Oracle® Application Server

Developer's Guide: JServlet Applications

Release 4.0.8.1

September 1999

Part No. A73043-01

ORACLE®

Oracle Application Server Release 4.0.8.1 Developer's Guide: JServlet Applications

Part No. A73043-01

Primary Author: Sanjay Singh

Contributors: Zhou Ye, Sumathi Gopalakrishnan, Ramani Jagedeba, Jerry Bortved, Jun Wang, Alice Chan, Bo Stern, Ryan Bennett, Yongwen Xu, Yi Liu, Ayub Khan

Contents

Preface

Audience

This book is for developers who want to create Java Servlet-based Web applications for Oracle Application Server.

Assumptions

We assume that the JServlet developer is familiar with the Java Servlet API Specification and Java development.

The Oracle Application Server Documentation Set

This table lists the Oracle Application Server documentation set.

Title of Book	Part No.
Oracle Application Server 4.0.8 Documentation Set	A66971-03
Oracle Application Server Overview and Glossary	A60115-03
Oracle Application Server Installation Guide for Sun SPARC Solaris 2.x	A58755-03
Oracle Application Server Installation Guide for Windows NT	A58756-03
Oracle Application Server Administration Guide	A60172-03
Oracle Application Server Security Guide	A60116-03
Oracle Application Server Performance and Tuning Guide	A60120-03
Oracle Application Server Developer's Guide: PL/SQL and ODBC Applications	A66958-02
Oracle Application Server Developer's Guide: JServlet Applications	A73043-01
Oracle Application Server Developer's Guide: LiveHTML and Perl Applications	A66960-02

Title of Book	Part No.
Oracle Application Server Developer's Guide: EJB, ECO/Java and CORBA Applications	A69966-01
Oracle Application Server Developer's Guide: C++ CORBA Applications	A70039-01
Oracle Application Server PL/SQL Web Toolkit Reference	A60123-03
Oracle Application Server PL/SQL Web Toolkit Quick Reference	A60119-03
Oracle Application Server JServlet Toolkit Reference	A73045-01
Oracle Application Server JServlet Toolkit Quick Reference	A73044-01
Oracle Application Server Cartridge Management Framework	A58703-03
Oracle Application Server 4.0.8.1 Release Notes	A66106-04

Conventions

This table lists the typographical conventions used in this manual.

Convention	Example	Explanation
bold	**oas.h** **owsctl** **wrbcfg** **www.oracle.com**	Identifies file names, utilities, processes, and URLs
italics	*file1*	Identifies a variable in text; replace this place holder with a specific value or string.
angle brackets	`<filename>`	Identifies a variable in code; replace this place holder with a specific value or string.
courier	`owsctl start wrb`	Text to be entered exactly as it appears. Also used for functions.
square brackets	`[-c string]`	Identifies an optional item.
	`[on|off]`	Identifies a choice of optional items, each separated by a vertical bar (|), any one option can be specified.
braces	`{yes|no}`	Identifies a choice of mandatory items, each separated by a vertical bar (|).
ellipses	n,...	Indicates that the preceding item can be repeated any number of times.

The term "Oracle Server" refers to the database server product from Oracle Corporation.

The term **“oracle**” refers to an executable or account by that name.

The term “*oracle*” refers to the owner of the Oracle software.

Technical Support Information

Oracle Global Support can be reached at the following numbers:

- In the USA: **Telephone: 1.650.506.1500**
- In Europe: **Telephone: +44 1344 860160**
- In Asia-Pacific: **Telephone: +61. 3 9246 0400**

Please prepare the following information before you call, using this page as a check-list:

- ❏ your CSI number (if applicable) or full contact details, including any special project information
- ❏ the complete release numbers of the Oracle Application Server and associated products
- ❏ the operating system name and version number
- ❏ details of error codes and numbers and descriptions. Please write these down as they occur. They are critical in helping WWCS to quickly resolve your problem.
- ❏ a full description of the issue, including:
  - **What** - What happened? For example, the command used and its result.
  - **When** -When did it happen? For example, during peak system load, or after a certain command, or after an operating system upgrade.
  - **Where** -Where did it happen? For example, on a particular system or within a certain procedure or table.
  - **Extent** - What is the extent of the problem? For example, production system unavailable, or moderate impact but increasing with time, or minimal impact and stable.
- ❏ Keep copies of any trace files, core dumps, and redo log files recorded at or near the time of the incident. WWCS may need these to further investigate your problem. For a list of trace and log files, see “Configuration and Log Files” in the *Administration Guide.*

For installation-related problems, please have the following additional information available:

- ❏ listings of the contents of $ORACLE_HOME (Unix) or %ORACLE_HOME% (NT) and any staging area, if used.
- ❏ installation logs (**install.log**, **sql.log**, **make.log**, and **os.log**) typically stored in the **$ORACLE_HOME/orainst** (Unix) or **%ORACLE_HOME%\orainst** (NT) directory.

Documentation Sales and Client Relations

In the United States:

- To order hardcopy documentation, call Documentation Sales: **1.800.252.0303.**
- For shipping inquiries, product exchanges, or returns, call Client Relations: **1.650.506.1500.**

In the United Kingdom:

- To order hardcopy documentation, call Oracle Direct Response: **+44 990 332200.**
- For shipping inquiries and upgrade requests, call Customer Relations: **+44 990 622300.**

Reader's Comment Form

Oracle Application Server 4.0 Developer's Guide: JServlet Applications

Part No. A73043-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have suggestions for improvement, please indicate the topic, chapter, and page number below:

Please send your comments to:

Oracle Application Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065

If you would like a reply, please provide your name, address, and telephone number below:

Thank you for helping us improve our documentation.

1
Introduction

The JServlet cartridge contains a Java Virtual Machine and Java class libraries. It provides a runtime environment for server-side Java applications written with the Java Servlet API Specification (available from http://java.sun.com).

The advantages of using the JServlet cartridge include:

- The JServlet cartridge provides better performance than CGI because there is no start-up and shutdown of the Java Virtual Machine required for each request. This also allows database connections to remain open for a session reducing the overhead of reconnecting to the database with each request.
- The JServlet cartridge takes advantage of Oracle Application Server's load-balancing, scalability, monitoring, logging and session management capabilities.
- The JServlet cartridge minimizes the use of system resources by running multiple JServlet cartridges on the same virtual machine when they belong to the same application. Free instances of applications are also used when available instead of creating new instances.
- The JServlet cartridge comes with the JServlet Toolkit, which provides a set of classes that can be used to generate HTML and access databases.

JServlets are run on the server side. The cartridge is not involved with running Java applets, which are downloaded and run on the client's machine.

Before invoking a servlet application through the JServlet cartridge, you provide the cartridge with configuration information such as a virtual path, environment variables and authentication information. This configuration is done with the Oracle Application Server Manager.

For general information about Java Servlets, refer to the Java Servlet API Specification. This is available from http://java.sun.com.

Contents

JServlet Terminology

In this book some terms are used frequently to describe components of Oracle Application Server. They include:

Table 1–1 Terms used in this book

Term	Definition
cartridge	Code that executes application logic and configuration data that defines its runtime behavior.
application	A collection of cartridges that share the same runtime environment.
instance	A JServlet instance.
JServlet runner	A servlet engine.
cartridge server	A process that all of the JServlet runners for an application run in. For JServlet cartridges, this is the Java Virtual Machine running on the server.

Developing Servlets for the JServlet Cartridge

This section is an introduction to developing servlets for the JServlet cartridge. To be able to develop such servlets, you will need to know the following:

Tools Required

The JServlet cartridge is a runtime environment, and as such, it does not come with any development tools. To develop servlets for the JServlet cartridge, you will need a compiler, a debugger, and other tools needed for servlet development. Oracle's JDeveloper and Sun's JDK and JSDK are tools that can be used when developing servlets.

Development Strategy

Because the JServlet cartridge is a runtime environment, it does not have built-in debugging facilities other than using print statements to generate messages to a log file.

Note: The JServlet cartridge redirects the System.out stream to the client's browser.

You should build and debug as much of your application as possible outside of the application server. Then, adding the JServlet Toolkit classes, you can finish the application. This allows you to make full use of the debugging tools in your development tool for the standard classes of your application.

The init() Method

The `init()` method of a servlet is called when a servlet is initialized. This method is only called once for each servlet. The servlet will not be able to handle any requests until its `init()` method has completed.

If any exception is thrown while executing init(), the instance will not be made available and all references to the servlet are released immediately. The JServlet cartridge will then return a failure on the request.

See "Initializing Servlets" on page 3-1 for more information about the `init()` method.

Entry Point Methods

The `service()` method dispatches requests to a servlet entry method. Depending on the type of request, this method is typically either the `doGet()` or `doPost()` method. Either one or both of these methods can be defined and used as an entry point into a servlet.

See the Java Servlet API Specification for more information on entry point methods.

The destroy() Method

When a JServlet application is being shut down, its `destroy()` method is called. No other method of the servlet will be executed before the `destroy()` method is called after the shutdown. This method will be called exactly once.

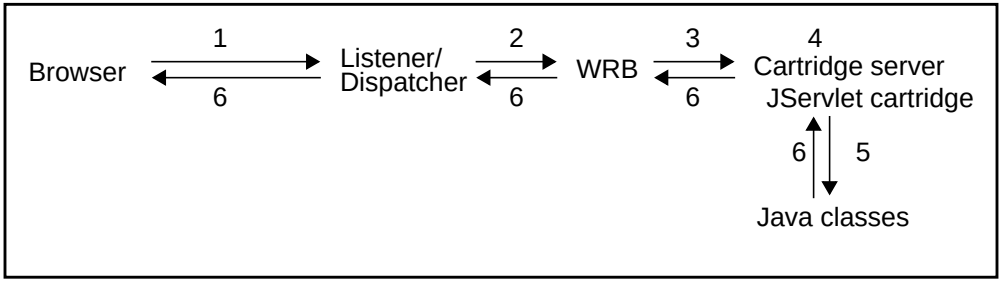
Although `destroy()` is not called until all instances have completed or timed out, service requests received while the destroy() method is waiting to be called will not be serviced.

See "Destroying JServlet Instances" on page 3-11 for more information about the `destroy()` method.

Control Flow

The following figure shows how Oracle Application Server handles a request for a JServlet cartridge.

Figure 1–1 Control flow for a JServlet cartridge



1. The listener component receives a request for a JServlet cartridge from a client. For example, a request for **http://karla-node/servlets/HelloServlet**.
2. The dispatcher sees that the request is for a cartridge and forwards the request to the WRB.
3. The WRB examines the URL and determines that the request is for a JServlet cartridge because the virtual path (**/servlets**) is mapped to a JServlet cartridge. The request is passed to the JServlet cartridge.
4. The JServlet cartridge running in a cartridge server process receives the request, examines the URL, and finds the name of the Java application (class) to

invoke. The name of the class is typically at the end of the URL. For example, in **/servlets/HelloServlet**, HelloServlet is the name of the class.

5. The JServlet cartridge loads the class and invokes its entry point method.
6. The Java application generates a response, including both the HTTP response header and response body, and returns it through a special output stream (HtmlStream). The JServlet cartridge receives the response and returns it to the WRB. The WRB forwards the response to the browser that invoked the request.

Versions of Java Supported

The JServlet cartridge supports Java 1.1.6 and the Java Servlet API Specification version 2.1. You should use a Java Development Kit (JDK) or an integrated development environment (IDE) based on these versions for developing applications for the JServlet cartridge.

Unsupported Java Servlet Features

The JServlet cartridge does not provide support for the following features from the Java Servlet API Specification.

- servlet chaining
- Java Server Pages (JSPs)

2

Tutorial

This tutorial provides step-by-step instructions on how to create a JServlet application, and how to invoke the application from a browser. The JServlet application invokes a Java application that creates a simple HTML page which displays “Hello World” in the browser.

> **Note:** You should be able to log into the OAS Manager as the “admin” user in order to add applications to the server. See the *Administration Guide* for more information.

Creating a JServlet application involves the following steps:

1. Creating the Java Class Files

This servlet consists of a Java class that prints “Hello World” to the client’s browser.

1. Enter the following Java source code in your Java development environment and save it as **HelloServlet.java**.

Example 2–1 HelloServlet.java code

```
import java.io.* ;
import javax.servlet.* ;
import javax.servlet.http.* ;

public class HelloServlet extends HttpServlet {
    public void doGet (
        HttpServletRequest request,
        HttpServletResponse response
        ) throws ServletException, IOException {

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<HTML><BODY>");
        out.println("<H2>Hello World</H2>");
        out.println("</BODY></HTML>");
    } // doGet
} // HelloServlet
```

2. In your development environment, set the environment variables to those set in Table 2–1 and compile the source file.

Table 2–1 Environment variables for JServlet applications

Name	Value
JAVA_HOME	$ORAWEB_HOME/jdk
CLASSPATH	$JAVA_HOME/lib/classes.zip $ORACLE_HOME/ows/cartx/jweb/classes/jweb.jar $ORACLE_HOME/ows/cartx/jweb/classes/jservlet.jar $ORACLE_HOME/orb/4.0/classes/yoj.jar (Unix only) $ORACLE_HOME/orb/classes/yoj.jar (NT only) $ORAWEB_HOME/classes/cosnam.jar
LD_LIBRARY_PATH (Unix only)	$ORACLE_HOME/cartx/jweb/lib $JAVA_HOME/lib/sparc/native_threads (Solaris only)
PATH (Windows NT only)	$ORACLE_HOME/cartx/jweb/lib
THREADS_FLAG	native

If you are using Sun's JDK, the Java compiler is called **javac**. Make sure it is in your path and type:

```
prompt> javac HelloServlet.java
```

The compiler creates a Java bytecode file called **HelloServlet.class**.

3. Check that the code is running correctly by running the bytecode.

   This can be done with your development environment. For example, if you were using JDeveloper to create and edit your servlet code, you could also use JDeveloper to run and test your servlet. Consult your development environment's documentation for more information.

   The HTML source should look like:

```
Content-type:  text/html

<HTML><BODY>
<H2>Hello World</H2>
</BODY></HTML>
```

4. Copy the complied **HelloServlet.class** file to the **$ORAWEB_HOME/test** directory. If you use another directory please use that path in the next stage. Create the directory if it does not exist.

2. Creating a JServlet Application and Cartridge

You will need to log into the Oracle Application Server Manager as the "admin" user in order to perform these steps. See the *Administration Guide* for more information about the Oracle Application Server Manager.

1. Access the Oracle Application Server Welcome Page. See "Accessing the Welcome Page" in the *Administration Guide* for instructions.
2. Click the ⊞ next to a site name to display the components on the site. You should see "Oracle Application Server", "HTTP Listeners", and "Applications".
3. Click "Applications" to display the applications in the right frame. Do not click the ⊞ next to Applications because this will show a list of applications for the site in the left frame, instead of Applications in the right frame.
4. On the applications page in the right frame, press the Add button (). The Add Application dialog opens.

5. In the Add Application dialog, enter the following information into the fields.

For this field...	Enter this value...	Because...
Application Type	JServlet	This is the type of application you will be adding.
Configure Mode	Manually	This enables you to enter configuration data using dialog boxes. The other option, From File, would get the same configuration data from a registration file.

Click Apply.

This displays a second Add Application dialog.

6. In the second Add Application dialog, enter the following information into the fields.

For this field...	Enter this value...	Because...
Application Name	helloservlet	This is the name used to identify the application.
Display Name	Hello Servlet Application	This name is used in the administration forms.
Application Version	1.0	This is the version of the application you are adding.

Click Apply.

When you click Apply, you get a Success dialog box, which contains a button that enables you to add cartridges to the application.

7. In the Success dialog box, click the “Add Cartridge to this Application” button. This displays the Add JServlet Cartridge dialog.

8. In the Add JServlet Cartridge dialog enter the following information into the fields.

For this field...	Enter this value...	Because...
Cartridge Name	hellocart	This name is used to identify your JServlet cartridge in your helloworld application.
Display Name	Hello Servlet Cartridge	This name is used in the administration forms.
Virtual Path	/jservlets/test	This is the path clients will use to invoke the JServlet cartridge.
Physical Path	%ORAWEB_HOME%/test	This field specifies the directory that contains the Java class files used by the cartridge. If you used another path in step 1.4, use that path here.

Note: Variables within Oracle Application Server are always surrounded with percent signs regardless of the platform.

For example, to use ORACLE_HOME in a path, enter %ORACLE_HOME%/<*rest_of_path*>.

Click Apply.

Now, the application you just added, helloservlet, contains one JServlet cartridge.

3. Reloading Oracle Application Server

After reconfiguring Oracle Application Server, you have to reload the server so your changes can take effect. To reload Oracle Application Server:

1. In the Oracle Application Server Manager, select the website by clicking on the website name at the top of the navigation tree.
2. Then, press the reload button () in the right frame.

The Oracle Application Server manager will reload all of the Oracle Application Server processes, listeners and applications.

4. Invoking the JServlet Cartridge

You can invoke the JServlet cartridge by typing the following URL into your browser:

```
http://<host>:<port>/jservlets/test/HelloServlet
```

host and *port* identify a listener that knows about the cartridge. This can be any listener on the application server except for the Node Manager listener. For example, you could use the www listener which, by default, resides on port 80.

Another way of invoking the cartridge is from an HTML page. For example, you can create the following HTML page (call it **hello.html**) to invoke your JServlet cartridge. The HTML page contains a link that calls the cartridge URL.

The following is the code of **hello.html**.

```
<HTML>
<HEAD><TITLE>Hello</TITLE></HEAD>
<BODY>
<H1>My First JServlet</H1>
<P><A HREF="http://<host>:<port>/jservlets/test/HelloServlet">Run my Hello
World class</a>
</BODY>
</HTML>
```

hello.html should be saved in a directory that is mapped to a virtual directory for a listener. If you would like to add a virtual directory, see "Configuring Directory Mappings" in the *Administration Guide*.

For More Information...

This chapter has briefly discussed various aspects of Oracle Application Server administration. More information on these topics can be found in the *Administration Guide*.

Table 2–2 Administration Guide topics of interest

Topic	Administration Guide Chapter and Section
Adding and configuring applications	Application Administration ■ Adding and Configuring Applications
Adding and configuring cartridges	Cartridge and Component Administration ■ Managing Cartridges

Table 2–2 Administration Guide topics of interest

Topic	Administration Guide Chapter and Section
Listener configuration	Managing and Configuring HTTP Listeners ■ Configuring a Listener
Adding virtual directories	Managing and Configuring HTTP Listeners ■ Configuring a Listener

For information about the JServlet Toolkit and extensions to the Java Servlet API Specification, refer to the *JServlet Web Toolkit Reference*.

For information about writing application and cartridge registration files which automate adding applications and cartridges, refer to *Cartridge Management Framework*.

3

Developing JServlets

Contents

Initializing Servlets

JServlet cartridges are initialized when the JServlet runner is created. When the cartridge is initialized, it's `init()` method will be executed. This method is passed a `ServletConfig` object which contains environment and configuration information stored in name-value pairs. This is illustrated in Example 3–1.

In Oracle Application Server you can provide parameters under the Application's Java Environment form. These parameters can be accessed with the `ServletConfig.getInitParameter()` method provided in the Java Servlet API. This is illustrated in Example 3–2.

Example 3–1 Accessing initial parameters in a servlet

> **Note:** This example assumes that the following name-value pair is set in the Java Environment form:
> `Servlet.<servlet>.initArgs = user=karla,pass=secret`
>
> Further information on the Java Environment form can be found in the *Administration Guide*.

```
public void init (ServletConfig cfg) throws ServletException {
    // print the user name and password
    System.out.println("User name: " + cfg.getInitParameter("user"));
    System.out.println("Password:  " + cfg.getInitParameter("pass"));
}
```

Example 3–2 Accessing the server environment

```
public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    // print the name and port of the server
    System.out.println("Server name: " + req.getServerName());
    System.out.println("Server port: " + req.getServerPort());
}
```

Accessing HTTP Request Information

Servlets are capable of receiving two types of information:

- Information from the server that is accessed through the `HttpServletRequest` object. This includes header, session and cookie information. This is illustrated in Example 3–3.
- Request parameters which are passed from the client to the servlet. This information is typically sent as name-value pairs using either the POST or GET method. These values are accessed using the `HttpServletRequest.getParameter()` method. This is illustrated later in Example 3–5.

Example 3–3 Accessing information from the HttpServletRequest object

```
public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    // print the user agent and remote user name
    System.out.println("User agent: " + req.getHeader("User-Agent"));
    System.out.println("User name:  " + req.getRemoteUser());
}
```

POST and GET Methods

POST and GET methods in the HTTP protocol instruct browsers how to pass parameter data (usually in the form of name-value pairs) to applications. The parameter data are usually generated by HTML forms.

Applications in the application server can use either method. The method that you use is as secure as the underlying transport protocol (HTTP or S-HTTP).

When you use the POST method, parameters are passed in the request body, and when you use the GET method, parameters are passed using the query string. These methods are described in the HTTP 1.1 specification, which is available at the W3C web site, http://www.w3c.org.

The limitation of the GET method is that the length of the value in a name-value pair cannot exceed the maximum length for the value of an environment variable, as imposed by the underlying operating system. In addition, operating systems have a limit on how many environment variables you can define.

Generally, if you are passing large amounts of parameter data to the server, you should use the POST method.

Depending on which method you choose to pass information to your servlet, it's entry point will be different. Servlets receiving information from the GET method, will enter at the `doGet()` method. Similarly, servlets receiving POST data will enter at the `doPost()` method.

Accessing Information from a Form

To pass information from a form to your servlet, the form needs to use either the POST or GET method to pass the data to the servlet. Example 3–4 shows an HTML file that can pass a name-value pair to a servlet called `formServlet` using the GET method.

Example 3–4 HTML page that passes form data to a servlet

```
<HTML>
<HEAD><TITLE>Forms and Servlets</TITLE></HEAD>
<BODY>
   <FORM METHOD="GET" ACTION="/jservlets/formServlet">
      <P>What is your name: <INPUT TYPE=TEXT NAME="name" SIZE=8>
   </FORM>
</HTML>
```

Example 3–5 shows the servlet code that uses the `getParameter()` method to retrieve the value passed in.

Example 3–5 Servlet to retrieve form data

```
public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    // print the name
    System.out.println("Hello " + req.getParameter("name"));
}
```

Generating HTTP Response Information

When a servlet responds to a HTTP request, the response object should include the HTTP response headers and the generated HTML page. The headers describe the properties of the response, such as the content type or the character set of the HTML page. The generated information will be written back to the client using an output stream.

The following topics are covered in this section:

- Output Streams
- Generating HTTP Response Headers
- Generating HTML
- Extending the oracle.html Package
- Closing Streams in the System Class

Output Streams

You can use one of three output streams with the JServlet cartridge. They are:

- HttpServletResponse stream — the standard servlet response object
- HtmlStream stream — the oracle.html package's output stream
- System.out stream — the standard java output stream

All three streams function equally well and are all thread safe in the Oracle Application Server context. You should develop applications using the stream you are most comfortable with, but never use different streams in a single application. Synchronization is not performed between the different streams so an application writing to both the `HtmlStream` and System.out streams would not be thread safe and could produce unexpected results.

> **Note:** The print methods in the `oracle.html` package write to the `HtmlStream` stream by default.

Generating HTTP Response Headers

To generate HTTP response headers:

- Use the `setContentType()` method to generate the first header line. More header information can be added by using the appropriate `HttpServletResponse` methods.

  ```
  Content-type: text/html
  ```

- Use the `println()` method to generate the other header lines as required.

All headers must be generated before you generate the content of your response. An empty line will separate the headers from the content.

Example 3–6 Generating HTTP response headers using setContentType()

```
public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    // generate the MIME type and character set header
    res.setContentType("text/html; charset=us-ascii");

    // rest of code goes here
    ...
}
```

Generating HTML

To generate HTML pages:

- Write the HTML code directly to the response object.
- Use the oracle.html package to generate your HTML page. This method writes its output to the HtmlStream.

While both methods work equally well, you should only use one method in an application. Writing to both the response object and the HtmlStream can have unexpected results. When both are used, the code is not thread safe.

Writing to the Response Object

To write to the response object use the `HttpServletResponse.getWriter.println()` method. Output written to this object will be sent to the client browser.

When writing directly to the response object you must also write any necessary header information.

Example 3–7 Writing a HTML page to the response object

```
public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    // generate the MIME type and character set header
    res.setContentType("text/html; charset=us-ascii");

    // generate the HTML page
    Printwriter out = res.getWriter();
    out.println("<HTML>");
    out.println("<HEAD><TITLE>Test Page</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("<H2>Sample text...</H2>");
    out.println("This is a sentence.");
    out.println("</BODY>");
    out.println("</HTML>");
}
```

Writing to the HtmlStream

The JServlet Toolkit includes the `oracle.html` package. This package can be used to create HTML pages quickly and elegantly.

To use the classes in the `oracle.html` package:

1. Create an HtmlPage object to contain the HEAD and BODY sections of your HTML page.

   ```
   HtmlPage htmlpage = new HtmlPage();
   ```

   You can specify the <TITLE> of your page in the `HtmlPage` constructor, or you can specify it later when you create the `HtmlHead` object.

2. If you want to add tags to the <HEAD> area, you need an `HtmlHead` object. You can get this in one of two ways:

   - Create the `HtmlHead` object and set it to the HtmlPage object:

     ```
     HtmlHead htmlhead = new HtmlHead();
     htmlhead.setBase("http://www.newbase.com"); // sets the BASE attribute
     htmlpage.setHead(htmlhead);                 // binds htmlhead to htmlpage
     ```

   - Get the HtmlHead object from the `HtmlPage` object:

     ```
     htmlpage.getHead().setBase("http://www.newbase.com");
     ```

3. Get the `HtmlBody` object for the page so that you can add content to the <BODY>. As with the <HEAD> area, you can get the `HtmlBody` in two ways:

   - Create the `HtmlBody` object:

     ```
     HtmlBody htmlbody = new HtmlBody();
     htmlpage.setBody(htmlbody);
     ```

   - Get the object from the `HtmlPage` object:

     ```
     htmlbody = htmlpage.getBody();
     ```

4. Add tags to the <BODY> section of the HTML page using classes in the `oracle.html` package. For example, the following line adds an <H1> heading to the page:

   ```
   htmlbody.Heading(1, "The first heading");
   ```

   See the *JServlet Toolkit Reference* for a complete class listing and usage examples of the `oracle.html` package.

Example 3–8 HTML page generation with the oracle.html package

```
public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    // STEP 1. Create an HtmlPage object
    HtmlPage hp = new HtmlPage();

    // STEP 2. Add a head to the HTML page
    hp.getHead().setTitle("HTML Example”);

    // STEP 3. Get the body object from hp
    HtmlBody hb = hp.getBody();

    // STEP 4. Add a string object ("Hello World!") to the page
    hp.getBody().addItem("Hello World!");

    // STEP 5. Print the header and contents to the output stream.
    hp.printHeader(); // prints the header information
    hp.print();       // prints the HTML page
}
```

This generates the following:

```
Content-type:  text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<!--  Generated by Oracle's Dynamic HTML Generation Package  -->
<HTML>
<HEAD>
<TITLE>Hello World</TITLE>
</HEAD>
<BODY>
Hello World!</BODY>
</HTML>
```

Extending the oracle.html Package

You can create your own HTML components by deriving them from the `CompoundItem` or `Container` classes. You can create high-level HTML classes that define particular layout styles and use them as templates. For example, you can create a `CompanyBanner` class that has a company logo, a hyperlink to its home page, and another hyperlink to its copyright notice. Each time when you want to include a

company banner, you create a `CompanyBanner` object, and fill in the company logo GIF file and the two URLs for the hyperlinks.

Example 3–9 Sample CompanyBanner class

```
class CompanyBanner extends CompoundItem {
    // Constructor takes an image and 2 links as arguments
    public CompanyBanner (String logoGIF,
                          String homepageLink,
                          String copyrightLink) {
        addItem(new Link(homepageLink,
        new Image(logoGIF, "Company Logo", IVAlign.TOP, true)));
        addItem(new Link(copyrightLink, "Copyright Notice"));
    } // constructor
} // CompanyBanner class
```

Then, you can simply add a `CompanyBanner` to your source code to generate HTML:

```
// Add a company banner
bd.addItem(new CompanyBanner("img/oracle.gif", "http://www.oracle.com",
                             "http://www.oracle.com/copyright.html");
```

You can also create HTML classes that encompass computation logic. For example, you can create a `BalanceSheet` class that performs a query of a customer's purchase information from a database and formats the results in HTML. To create a balance sheet for a customer, simply instantiate a `BalanceSheet` object and specify the customer's identifier. Then add the `BalanceSheet` item to your `HtmlPage`.

Closing Streams in the System Class

You must not close streams from the `System` class in your servlet because the JServlet cartridge redirects the output of the `System.out` and `System.err` streams. If you close the streams, the output from the stream can appear as binary data (if any) to the client. Instead of closing the stream, call the `flush()` method to ensure that the contents of the stream are written.

Developers should note that it is possible to inadvertently close a `System` stream. Example 3–10 demonstrates this.

Example 3–10 Inadvertently closing the system stream

```
OutputStreamWriter osw = new OutputStreamWriter(System.out);
osw.write('my message');
osw.close; // Closes the System.out stream. Don't do this!
osw.flush; // Do this instead.
```

Invoking JServlet Cartridges

To invoke a JServlet cartridge, the URL must be in the following format:

`http://`*`hostname`*`[:`*`port`*`]/`*`virtual_path/java_class_name`*`[`*`path_info`*`][?`*`QUERY_STRING`*`]`

where:

- *hostname* specifies the machine where the application server is running.
- *port* specifies the port at which the application server is listening. If omitted, port 80 is assumed.
- *virtual_path* specifies a virtual path mapped to the JServlet cartridge.
- *java_class_name* specifies the Java class to run. This class must contain an entry point method. If a query string is included in the URL then there must be a `doGet()` method.
- *path_info* provides additional path information for the servlet. The path can be accessed from the servlet using the `HttpServletRequest.getPathInfo()` method.
- *QUERY_STRING* specifies parameters (if any) for the Java class. The string follows the format of the GET method. For example, multiple parameters are separated with the & character and spaces are replaced with the + character. If you use HTML forms to generate the string (as opposed to generating the string yourself), the formatting is done automatically for you.

For example, if a browser sends the following URL:

```
http://karla:2525/games/addPlayer/template/player.html?name=Steve+Kerr&pos=guard
```

then the application server running on **karla** and listening at port **2525** would handle the request. When the listener receives the request, it passes the request to the WRB because it sees that the **/games** virtual path is configured to call a JServlet cartridge. The WRB sends the request to a cartridge server that is running the cartridge.

The JServlet runner searches the directories, jar, and zip files in its physical path associated with the virtual path and then the CLASSPATH for the **`addPlayer`** class specified in the URL. This class uses the additional path and query information. The class' `doGet()` method is executed since the information is sent in the URL. The doGet() method will need to call the `getPathInfo()` method to use find the **template/player.html** file.

The ***name*** parameter gets the value "Steve Kerr", the + is replaced by a space character before the class can use it. The ***pos*** parameter gets the value "guard".

To use the values of the parameters in your Java class, use the `getParameter()` method in the `HttpServletRequest` class.

Example 3–11 Getting parameters from a URL

```
public class addPlayer extends HttpServlet {
    public void doGet (HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        // get the parameters
        String player = req.getParameter("name");
        String position = req.getParameter("pos");

        // do something with the parameters
        ...
    } // doGet
} // addPlayer
```

Destroying JServlet Instances

A servlet's `destroy()` method will only be called when the cartridge server is stopped or shut down. The Oracle Application Server servlet engine guarantees that no other methods of a servlet are being executed before the destroy method is called.

Although the `destroy()` method is not called until all pending requests for a servlet have completed or timed out, all new requests for the to-be-destroyed instance will be refused.

Runtime Interpreter Options

In addition to the application and cartridge configuration parameters, you can specify the runtime options of Sun's Java interpreter in a Java cartridge environment.

Note: The basic cartridge configuration options can be found in "Cartridge and Component Administration" in the *Administration Guide*.

To specify these options for the cartridge, you use the Java Environment form. The options you specify apply to the entire application, but you can set different options for different applications.

Table 3–1 lists the names and valid values of these options. This is a complete list and no other runtime options are supported.

Table 3–1 Mapping Java interpreter options to cartridge configuration flags

Sun's Java interpreter option	Option name in the Java Environment form	Value
-verbose	VERBOSE	`true` or `false`
-noasyncgc	NOASYNCGC	`true` or `false`
-verbosegc	VERBOSEGC	`true` or `false`
-checksource	CHECKSOURCE	`true` or `false`
-ss	C_STACK	A number followed by M, m, K, or k, where M or m indicates megabytes, and K or k indicates kilobytes.
-oss	JAVA_STACK	A number followed by M, m, K, or k, where M or m indicates megabytes, and K or k indicates kilobytes.
-ms	INITIAL_HEAP	A number followed by M, m, K, or k, where M or m indicates megabytes, and K or k indicates kilobytes.
-mx	MAX_HEAP	A number followed by M, m, K, or k, where M or m indicates megabytes, and K or k indicates kilobytes.
-verify	VERIFY	`true` or `false`
-verifyremote	VERIFYREMOTE	`true` or `false`

Table 3–1 Mapping Java interpreter options to cartridge configuration flags

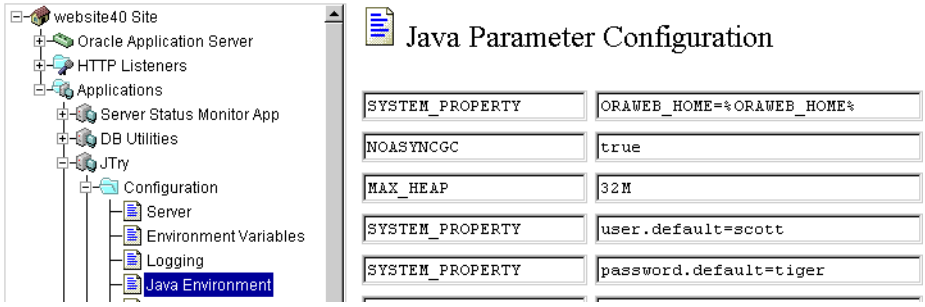
Sun's Java interpreter option	Option name in the Java Environment form	Value
-noverify	NOVERIFY	`true` or `false`
	JAVA_COMPILER	`symcjit` (for Windows NT) `sunwjit` (for Solaris)
-D	SYSTEM_PROPERTY	*prop=value*

Note the following:

- Sun's "-debug" option is not supported.
- Multiple "SYSTEM_PROPERTY" options can be used.
- "true / false" values are case-sensitive.

For example, if you want to turn off the asynchronous garbage collector, set maximum heap size to 32 Mbytes, and define system properties `user.default` and `password.default`, the Java Environment form for your application would look like the following:

Figure 3–1 Example Java Environment form



The options and their values are written to the `[APPLICATION.<appName>.JAVA]` section of the **wrb.app** file:

```
[APPLICATION.myApp.JAVA]
NOASYNCGC        = true
MAX_HEAP         = 32M
SYSTEM_PROPERTY = user.default=scott
SYSTEM_PROPERTY = password.default=tiger
```

4

Advanced JServlet Programming

This chapter will discuss advanced programming topics for the JServlet cartridge.

Contents

Using Sessions

Since servlets are inherently stateless and serially reusable, clients' state information cannot be stored directly in the servlet. To provide this functionality, JServlet developer's can implement the HttpSession package with some extensions. The session API allows developer's to preserve state information across requests to the servlet.

Sessions allow you to associate information with a particular user. This means that the same application can be used in multiple instances without losing data integrity. Using sessions guarantees that the next request from an application will go to the proper instance. The JServlet cartridge supports sessions through a programming interface. These *programmable sessions* store their state information independently from the servlet instance.

Implementing programmable sessions is similar to the implementing a Java Servlet with the Java Servlet API Specification. However, the JServlet cartridge does provide some additional functionality with the distributed session model.

Programmable Sessions

Programmable sessions are identified by an internally generated and globally unique string. This string is created when a session is begun and is then subsequently used by the cartridge runtime and the application to identify a session. This session identifier is passed to the client as part of a browser cookie or if a cookie is not available, by URL rewrite.

Using the `oracle.OAS.servlet.http.HttpSession package`, sessions are managed by the cartridge runtime using cache management system to save and restore session information. All sessions will have a "maximum idle time" associated with them. This is the maximum time allowed between accesses to a session. If this time is exceeded, the session and its associated state will be removed from the cache. Future attempts to access an expired session will generate an exception.

The maximum idle time can be set by:

- the application using the `setMaxInactiveInterval` method.
- the OAS Manager using the application's Web Parameters form. See the *Administration Guide* for more information.

If the setMaxInactiveInterval method is not used in the application, the value in the Web Parameters form will be used. Both methods use a default value of 600 seconds (10 minutes).

An application will have direct access to state objects stored within a session using the session class methods `getValue`, `putValue`, and `removeValue`. The session manager views all state objects as Java Objects, so the structure and content of the state stored within a session can be defined by the application. State objects are identified by application defined names. Calling `putValue` on an existing object in the session will replace it with a new value. Only the latest version of a particular object will be saved. Access to a session is serialized so only one user instance may access a session and the session objects at a time.

Environment Variables

In addition to environment variable settings specified in Table 2–1, "Environment variables for JServlet applications" on page 2-2, add the following file to your CLASSPATH when using sessions.

- $ORACLE_HOME/orb/4.0/classes/session.jar (Unix only)
- $ORACLE_HOME/orb/classes/session.jar (NT only)

Example 4–1 Creating a JServlet session

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionServlet extends HttpServlet {

  public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    HttpSession session;
    // then write the data of the response
    PrintWriter out = res.getWriter();
    String UseCookie  = req.getParameter("usecook");
    oracle.OAS.servlet.ServletRequest req1 =
       (oracle.OAS.servlet.ServletRequest) req;

    String Invalid = req.getParameter("invalid");
    String NoArg   = req.getParameter("noarg");
    String LocalSess  = req.getParameter("local");
    String distSess   = req.getParameter("distrib");

    if (Invalid == null)   Invalid = "false";
    if (NoArg == null)     NoArg = "false";
    if (distSess == null)  distSess = "false";

    // Create or get the session object
    if (distSess.equals("true")) {
       session = req1.getSession(true,true,false);
    } else {
       if (NoArg.equals("true"))
          session = req.getSession();
       else
          session = req.getSession(true);
    } // if-else

    if (UseCookie != null) {
       if (UseCookie.equals("true"))
          req1.useCookie(true);
```

```
   else
      req1.useCookie(false);
} // if

// set content type and other response header fields first
res.setContentType("text/html");

out.println("<HEAD><TITLE> "+"SessionServlet Output " +
            "</TITLE></HEAD><BODY>");
out.println("<h1> SessionServlet Output </h1>");

if (Invalid.equals("true")) {
   session.invalidate();

   session = req.getSession(false);
   if ( session == null )
      out.println("req.getSession(false): PASSED<br>");
   else
      out.println("req.getSession(false): FAILED<br>");

   if (NoArg.equals("true"))
      session = req.getSession();
   else
      session = req.getSession(true);

   if (UseCookie != null) {
      if (UseCookie.equals("true"))
         req1.useCookie(true);
      else
         req1.useCookie(false);
   } // if UseCookie != null
} // if Invalid.equals("true")

oracle.OAS.servlet.http.HttpSession session1 =
                  (oracle.OAS.servlet.http.HttpSession) session;

if (LocalSess != null) {
   if (LocalSess.equals("true"))
      session1.localSession(true);
   else
      session1.localSession(false);
} // if LocalSess != null

Integer ival = (Integer) session.getValue("sessiontest.counter");
if (ival==null)
```

```
   ival = new Integer(1);
else
   ival = new Integer(ival.intValue() + 1);
session.putValue("sessiontest.counter", ival);

// Expire the session after 3 minutes Idle period
session.setMaxInactiveInterval(180);
out.println("You have hit this page <b>" + ival + "</b> times.<p>");
out.println("Click <a href=" +
    res.encodeUrl(HttpUtils.getRequestURL(req).toString()) + ">here</a>");
out.println(" to ensure that session tracking is working even " +
    "if cookies aren't supported.<br>");
out.println("Note that by default URL rewriting is not enabled" +
    "due to it's expensive overhead");

out.println("<h3>Request and Session Data:</h3>");
out.println("Session ID in Request: " +
    req.getRequestedSessionId());
out.println("<br>Session ID in Request from Cookie: " +
    req.isRequestedSessionIdFromCookie());
out.println("<br>Session ID in Request from URL: " +
    req.isRequestedSessionIdFromUrl());
out.println("<br>Valid Session ID: " +
    req.isRequestedSessionIdValid());
out.println("<br>Local Session: " +
    session1.isLocalSession());
out.println("<h3>Session Data:</h3>");
out.println("New Session: " + session.isNew());
out.println("<br>Session ID: " + session.getId());
out.println("<br>Creation Time: " + session.getCreationTime());
out.println("<I>(" + new Date(session.getCreationTime()) + ")</I>");
out.println("<br>Last Accessed Time: " +
    session.getLastAccessedTime());
out.println("<I>(" + new Date(session.getLastAccessedTime()) + ")</I>");

try {
    out.println("<br>Max Inactive Interval: " +
        session.getMaxInactiveInterval() + " seconds");
} catch (Exception e) {
    out.println("<br>Max Inactive Interval: Invalid Session");
} // try

HttpSessionContext context = session.getSessionContext();

if (context != null) {
```

```
        out.println("<h3>Session Context Data:</h3>");

        for (Enumeration e = context.getIds(); e.hasMoreElements() ;) {
            out.println("Valid Session: " + (String)e.nextElement()+ "<br>");
        } // for
      } // if context != null

      // After 5 times invalidate the session
      if (ival.intValue() == 5) {
        out.println("<br><br><b>" +
            "Already accessed 5 times, invalidating the session."+"<b>");
        session.invalidate();
      } // if
      out.println("</BODY>");
      out.close();
    } // doGet()

    public String getServletInfo() {
      return "A Session Servlet";
    }
} // SessionServlet class
```

Local and Distributed and Sessions

The two models for creating sessions with the JServlet cartridge are local and distributed. They have the same functionality, but each offers advantages over the other. You can choose the model that is best suited for your application. By default, local sessions are created.

Local sessions store state information within the process running the servlet. This provides for quick retrieving and updating of state information. As a result, applications created under this model will perform more efficiently.

To create a distributed session, use the `getSession(create, distributed, secure)` method where distributed is `true.For example,`

```
oracle.OAS.servlet.http.HttpSession sess =
     oracle.OAS.servlet.ServletRequest.getSession(true, true, false)
```

Sessions started under secure sessions will be based on the local session model to preserve the security of the state. To create a secure connection, use the `HttpSession.getSession(create, distributed, secure)` method where distributed is `false` and secure are `true`. Typically, local sessions spool

state objects to the server's disk, however, secure sessions will not write any information to the disk.

In the distributed session model, the application's state is replicated to other processes and / or nodes. If the original process should die for some reason, session information will automatically be propagated to another instance while maintaining the original session identifier. This adds a high level of fault tolerance to your JServlet applications. To allow this propagation, session objects must implement the `java.io.Serializable` interface.

When using distributed sessions, the following packages must be imported:

- oracle.OAS.servlet.http.HttpSession
- oracle.OAS.servlet.ServletRequest.

> **Note:** If the `java.io.Serializable` interface is not implemented, the session will be marked as a local session. This overrides all methods of creating a distributed session.

Binding Session Objects

Session objects can perform actions when they are bound (added) or unbound (removed) from a session. Using the `HttpSessionBindingListener` interface, objects can be notified as they are bound and unbound. This also marks the session as a local session.

After implementing the `HttpSessionBindingListener` interface, you must implement its `valueBound()` and `valueUnbound()` methods. These methods can be used to coordinate shared resources and begin and end transaction services.

When a servlet that implements the interface calls the HttpSession.putValue() method, the `valueBound()` method will be called. The valueUnbound() method will be called for a bound object when:

- calling removeValue()
- timing out the session
- invalidating the session.

Example 4–2 Binding session events

```
public class SessionBindTest extends HttpServlet {
    static PrintWriter out;
```

```
    public void doGet ( HttpServletRequest req, HttpServletResponse res )
                    throws ServletException, IOException {
        // Get the current session object, create one if necessary
        HttpSession session = req.getSession(true);

        // set content type and other response header fields first
        res.setContentType("text/html");

        // then write the data of the response
        out = res.getWriter();

        out.println("<HTML>");
        out.println("<HEAD><TITLE>Session Binding Event Test</TITLE></HEAD>");
        out.println("<BODY>");

        // Add a custom listener
        session.putValue("bindings.listener",
                         new CustomBindingListener(getServletContext()));
        // out.println("<br>This page intentionally left blank");
        out.println("</BODY>");
        out.close();
    } // doGet

    public String getServletInfo() {
        return "A Servlet to test Session Binding Events.";
    } // getServletInfo
} // SessionBindTest class

class CustomBindingListener implements HttpSessionBindingListener {
   // Save a ServletContext to be used for its log() method
    ServletContext context;

    public CustomBindingListener( ServletContext context ) {
        this.context = context;
    }

    public void valueBound ( HttpSessionBindingEvent event ) {
        // log the current date and time
        context.log("[" + new Date().toString() + "]");
        String state = "BOUND   as " + event.getName();
        state += " to   " + event.getSession().getId();
        // log, print and save the current state
        context.log(state);
        SessionBindTest.out.println("<BR>" + state);
        saveState(state);
```

```
    } // valueBound

    public void valueUnbound( HttpSessionBindingEvent event ) {
        context.log("[" + new Date().toString() + "]");
        String state = "UNBOUND as " + event.getName();
        context.log(state);
        SessionBindTest.out.println("<BR>" + state);
        saveState(state);
    } // valueUnbound

    public void saveState(String state) {
      try {
        String log_file = "SessionBinding.log";
        FileWriter fileWriter = new FileWriter(log_file,true);
        fileWriter.write(new Date().toString() + "\n");
        fileWriter.write(state + "\n");
        fileWriter.close();
        return;
      } catch ( IOException e ) {
        System.err.println("Error saving session state.");
       }
    } // saveState
} // CustomBindingListener class
```

Changes from the JWeb Session Model

The JWeb cartridge supported configurable sessions. Only programmable sessions are supported by the JServlet cartridge. To migrate an existing JWeb application that uses sessions, convert the application to the programmable model completely.

Using class and static members to store state information for sessions is also not supported for the JServlet cartridge. The servlet session APIs should be used to store session state information.

Inter-Cartridge Exchange (ICX) Service

The JServlet Toolkit provides classes for accessing Oracle Application Server's Inter-cartridge Exchange service (ICX). ICX allows one cartridge to invoke another cartridge and retrieve output from it. With this communication service, you can produce applications with output from different cartridges, where each one is specialized in handling a certain type of request or producing a certain type of result. For example, you can write a Java application to search for the users logging

on to a system, then invoke a PL/SQL cartridge to generate a report of the users' information from the database and include it in the HTML output.

The `oracle.OAS.Services.ICX` package contains the following classes:

- `ICXRequest` — sends ICX requests
- `ICXResponse` — the response to ICX requests
- `ICXInitFailedException` — an exception that is thrown when creating a new `ICXRequest` fails
- `IncompatibleWithProtocolException` — an exception that is thrown when a wallet is used with the http scheme

`ICXRequest` contains methods that enable you to perform the following tasks:

Table 4–1 Methods in ICXRequest

To...	Use this method...
Set up an ICX request	`ICXRequest()`, the ICXRequest constructor
Specify the request method (for example, POST or GET)	`setMethod()`
Specify additional headers to include in the request	`setHeader()` or `setHeaders()`
Specify the contents of the request	`setContent()` or `setContents()`
Specify user and password information	`setAuthInfo()`
Use SSL with the ICX request	`setWalletInfo()`
Enable transactions	`enableTransaction()`
Disable transactions	`disableTransaction()`
Send the request	`connect()`

The sequence of calls that you would invoke to send an ICX request follows in Example 4–3.

Example 4–3 Sending an ICX request

```
// import packages
import oracle.owas.wrb.services.http.*;
import oracle.OAS.Services.ICX.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// set up the ICXRequest object
ICXRequest icxreq = new ICRequest("http://machine:port/path");

// set up properties for the request (for example, method, headers, content,
// security, transaction)
```

```
// send the request and get an ICXResponse object
ICXResponse response = icxreq.connect();
if (icxresp == null) {
   // if the response is null, then print an error message and return.
   System.err.println("null response");
   return;
}
```

Note that you must specify a fully qualified URL in the `ICXRequest` constructor. You have to specify the scheme (http or https), the server, the port number (if necessary), and the virtual path. You cannot specify just the virtual path.

The `connect()` method returns an `ICXResponse` object or null if an error occurred. You can then retrieve data from the `ICXResponse` object using the following methods:

Table 4–2 Methods in ICXResponse

To...	Use this method...	Return value
Get header information	`getHeader()` or `getHeaders()`	Object for `getHeader()` Hashtable for `getHeaders()`
Get content information	`getContent()`	InputStream
Get the status of the response	`getStatusCode()` and `getReasonPhrase()`	int for `getStatusCode()` String for `getReasonPhrase()`
Get the realm	`getRealm()`	String
Get the HTTP version	`getHTTPVersion()`	String
Determine if the request was sent via a proxy	`usingProxy()`	boolean

Note: You cannot create an `ICXResponse` object using `new`. You have to get the object from the return value of the `connect()` method.

Cookies

The header for ICXRequest is the same as the original header, except that cookies are not included. To include cookie information in the header, you have to call `setHeader("Cookie", string)` explicitly.

Setting Multiple Fields in Headers or Content

You can set multiple fields in the header or content using `setHeaders()` or `setContents()` methods. These methods use a `Hashtable` object to contain the fields. The keys in the hash table are the field names, and the values must be `String` or `Vector` objects. If the value is a `Vector` object, it must be a `Vector` of `Strings`.

Transactions

> **Note:** Transactions are available only in the Enterprise Edition of Oracle Application Server.

If the target of an ICX request is for a cartridge that uses the Transaction service, you have to enable the transactional attribute before you send the ICX request. To enable the transactional attribute, call the `enableTransaction()` method in the `ICXRequest` class.

```
icxreq = new ICXRequest(url); // create the request
icxreq.enableTransaction(); // enable transaction

// set up other attributes of the request
ICXResponse icxresp = icxreq.connect(); // send the request
```

SSL

To use SSL with ICX requests, the URL of the ICX request uses the https scheme, instead of the http scheme. The URL is specified in the ICXRequest constructor.

When using SSL with ICX, you need to associate the ICX request with a wallet. A wallet contains private keys, certificates, and trust points that SSL can use. To define and manage wallets, you use the Oracle Wallet Manager. See the *Security Guide* for details.

Use the `ICXRequest.setWalletInfo()` method to associate ICX requests with wallets. The syntax of the method is:

```
public void setWalletInfo(String walletloc, String password)
   throws IncompatibleWithProtocolException
```

The method takes two parameters: a wallet location and the wallet's password. The wallet location is specified using a WRL (wallet resource locator). A WRL has the following syntax:

```
<Wallet Type>:<Wallet Type Parameters>
```

- *wallet type* specifies that the wallet is stored in a flat file in the operating system. Its value is "`file`".
- *wallet type parameters* specify the location of the wallet. For wallets stored in a flat file, the wallet type parameter is a full path to the wallet.

The following WRLs specify that the wallet is stored in the **/home/wallets** directory or the **c:\wallets\new_projects** directory.

```
file:/home/wallets
```

```
file:c:\wallets\new_projects
```

The wallet's password is assigned when the wallet was created.

You must ensure that the scheme for ICX requests that use wallets is https, not http. Otherwise, the method throws the `IncompatibleWithProtocolException` exception. The scheme is specified in the ICXRequest constructor.

Exceptions and Errors

The `ICXInitFailedException` exception is thrown when an ICXRequest object cannot be created. The object cannot be created because of low memory and/or failure to initialize required WRB components such as the authentication server. If you get this exception, you cannot perform any ICX-related operations.

You should check the return value of `ICXRequest.connect()`. If an error occurs, the method returns null. In this case, you do not have an ICXResponse object; you should return an error message to the user and stop processing the ICX request.

The `IncompatibleWithProtocolException` exception is thrown by `ICXRequest.setWalletInfo()` when you associate a wallet with a non-SSL ICX request, that is, the scheme of the ICX request is http, instead of https.

Example (not using SSL)

The following example sends a request to a PL/SQL cartridge. It also sends a user ID and password to the cartridge:

Example 4–4 Sending a request to a PL/SQL cartridge without using SSL

```
import oracle.owas.wrb.services.*;
import oracle.owas.wrb.services.http.*;
import oracle.OAS.Services.ICX.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ProcessRequestFromBrowser extends HttpServlet {
   public void doPost(
      HttpServletRequest req,
      HttpServletResponse res ) throws ServletException, IOException {

      String vpath = "/hr/getEmp";
      String method = "POST";
      String user, password;
      ICXRequest icxreq = null;
      String machine = req.getServerName();
      String port = req.getServerPort();
      String url = "http://" + machine + ":" + port + vpath;

      // set up an ICX request
      try {
         icxreq = new ICXRequest(url);
      } catch (java.net.MalformedURLException e) {
         System.err.println("Invalid URL");
      } catch (ICXInitFailedException e) {
         // send appropriate error message to client browser; something like
         // "Unable to ccomplete your request at this time. Please try again."
         System.err.println("Unable to create an ICX request.");
      }

      // set the method for the ICX call to be POST
      try {
         icxreq.setMethod(method);
      } catch (java.net.ProtocolException e) {
         System.err.println("Unable to set method");
      }
```

```
      // pull out the username and password from the headers.
      // Retrieve "user" and "password" from the HTTP header.
      user = req.getParameterValue("user");
      password = req.getParameterValue("password");

      // set the username and password.
      icxreq.setAuthInfo(user, password);

      // send the ICX request. connect() returns an ICXResponse object.
      ICXResponse icxresp = icxreq.connect();
      if (icxresp==null) {
         // if the response is null, then print an error message and return.
         System.err.println("null response")
         return;
      }

      // get the status of the response
      int status = icxresp.getStatusCode();
      if (status >= 400) { // an error occurred
         System.err.println(icxresp.getReasonPhrase());
      } else {
         // ICX request successful, get data from the response
         InputStream contentStream = icxresp.getContent();
         // process the data using methods in the BufferedInputStream class
         ...
      } // if
   } // doPost()
} // class
```

Example (using SSL)

The following example sends a request to a PL/SQL cartridge using SSL.

Example 4–5 Sending a request to a PL/SQL cartridge using SSL

```
import oracle.owas.wrb.services.*;
import oracle.owas.wrb.services.http.*;
import oracle.OAS.Services.ICX.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
public class ProcessRequestFromBrowser_SSL extends HttpServlet {
   public static void doPost (
      HttpServletRequest req,
      HttpServletResponse res ) throws ServletException, IOException {

      String vpath = "/hr/getEmp";
      String method = "POST";
      ICXRequest icxreq = null;
      String machine = req.getServerName();
      String port = req.getServerPort();
      // the scheme has to be https to use wallets
      String url = "https://" + machine + ":" + port + vpath;
      String wallet = "file:/home/joe/myWallet/";
      String wallet_password = "topsecret";

      // set up an ICX request
      try {
         icxreq = new ICXRequest(url);
      } catch (java.net.MalformedURLException e) {
         System.err.println("Invalid URL");
         // send appropriate error message to client browser; something like
         // "Unable to ccomplete your request at this time. Please try again."
      } catch (ICXInitFailedException e) {
         System.err.println("Unable to create an ICX request");
      }

      // set the method for the ICX call to be POST
      try {
         icxreq.setMethod(method);
      } catch (java.net.ProtocolException e) {
         System.err.println("Unable to set method");
      }

      // set the wallet
      try {
         icxreq.setWalletInfo(wallet, wallet_password);
      } catch (IncompatibleWithProtocolException e) {
         System.err.println("Unable to set wallet");
      }
```

```
      // send the ICX request. connect() returns an ICXResponse object.
      ICXResponse icxresp = icxreq.connect();
      if (icxresp==null) {
         // if the response is null, then print an error message and return.
         System.err.println("null response")
         return;
      }

      // get the status of the response
      int status = icxresp.getStatusCode();
      if (status >= 400) { // an error occurred
         // call user-defined method to return error message to the browser
         // returnErrorMessage(icxresp.getreasonPhrase());
         System.err.println(icxresp.getReasonPhrase());
      } else {
         // ICX request successful
         // get data from the response
         InputStream contentStream = icxresp.getContent();
         // process the data using methods in the BufferedInputStream class
         ...
      } // if
   } // doPost()
} // class
```

Servlet Concurrency

In Oracle Application Server, a cartridge can run in multiple threads by adjusting the number of threads parameter in the cartridge configuration form. Also, adjusting the number of JServlet runners allows for the concurrent execution of your cartridge logic. See the *Administration Guide* for more information on cartridge configuration.

Oracle Application Server uses a thread pool to implement thread concurrency for the JServlet cartridge. The thread pool and its threads are also maintained and managed by Oracle Application Server. Because of this, applications cannot submit a thread to the pool or use an application created thread to service incoming requests. All incoming requests will be serviced by threads created by the JServlet runtime.

Thread Safety and the SingleThreadModel Interface

Servlets can indicate that they are thread safe to the JServlet cartridge by implementing (or not implementing) the `SingleThreadModel` interface. Servlets that implement this interface are assumed to have code that is thread safe on the class level. It is, therefore, the developer's responsibility to ensure that the code is thread safe. The servlet instance is assumed to not be thread safe. This means that Oracle Application Server will not reenter methods of a servlet instance, but can call methods of different servlet instances at the same time.

Objects Declared as Static

Regardless of the threading model, declaring objects as `static` will cause the objects to be shared by all threads in the process. Care should be exercised when declaring variables as static because all instances of a servlet can change the value of the variable.

Servlets That Implement SingleThreadModel

In order to achieve concurrency on servlets that implement `SingleThreadModel`, Oracle Application Server instantiates multiple servlet instances in one process. By having multiple instances and making simultaneous calls to the `service()` methods of the different instances, Oracle Application Server achieves concurrency.

Oracle Application Server guarantees that there will be enough servlet instances to service simultaneous requests for these servlets. These instances will be created on demand as requests are received. As instances become free, they will be reused when possible. These instances will also be kept alive for the lifetime of the cartridge instance once instantiated.

Servlets That Do Not Implement SingleThreadModel

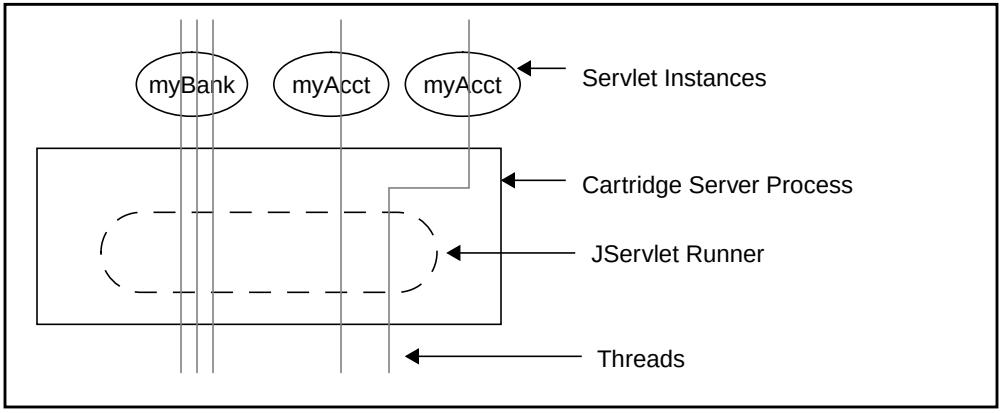
For servlets that do not implement the SingleThreadModel, Oracle Application Server instantiates exactly one servlet instance per cartridge process and makes calls to the instance upon receiving each request. These servlets are assumed to be fully thread safe.

These instances will be threaded if there are simultaneous requests for the cartridge. A servlet instance will be instantiated when the first request for the servlet is loaded into the cartridge server process. The servlet instances stay alive for the lifetime of the JServlet runner and are reused for any subsequent requests for the same servlet.

A JServlet Threading Scenario

Figure 4–1 illustrates the relationship between the various threading components described earlier in this section.

Figure 4–1 The relationship between threading components



In this figure, myBank is an instance of a servlet that does not implement SingleThreadModel. The myAcct instances are servlet instances of a servlet that does implement the single thread model. Currently, myBank is handling three requests and myAcct servlet is handling two.

Because myBank does not implement the model, it is assumed to be fully thread safe. Therefore, it is safe to process multiple requests through the instances. If another request comes in for myBank's servlet before the three current requests finish, a fourth thread will be assigned to this instance.

The servlet for myAcct implements the `SingleThreadModel` interface and therefore the instance is not thread safe. (Note, only the instance for myAcct is not thread safe, the code is assumed to be thread safe at the class level.) For each request to this servlet, a new instance is required. If another request for myAcct's servlet occurs before the two current requests complete, a new instance will be created and a new thread will be assigned to the new instance.

Changes from the JWeb Threading Model

Since each JWeb cartridge instance had its own class loader, multithreading could be achieved with code that was not thread safe. The JServlet cartridge only uses one class loader for each JServlet runner and therefore, requires that the JServlet code is thread safe.

Spawning Sub-Threads

> **Note:** Spawning sub-threads is not recommended because it limits Oracle Application Server's load balancing and management capabilities.

You must follow these guidelines when spawning sub-threads:

- Each thread must be associated with an instance of the oracle.owas.wrb.`WRBRunnable` class. This class enables the thread to use classes in the oracle.html package and also `System.out` for generating HTML output.
- The entry point thread, the thread running the entry method — `doGet()` or `doPost()` — must wait for the sub-threads to finish before it returns. Some synchronization may be necessary to accomplish this.

You must follow these guidelines because the servlet engine within the JServlet cartridge can run multiple servlet instances simultaneously. As a result, it needs to be able to associate user-created threads with the correct clients.

Example 4–6 creates four threads, and each thread prints its name to the HTML output using classes from the `oracle.html` package. Note that the `HtmlPage` is passed to the thread.

Example 4–6 Creating and adopting sub-threads

```
import oracle.owas.wrb.*;
import oracle.html.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```
public class HelloThread extends HttpServlet {
    public void doPost( HttpServletRequest req,
                        HttpServletResponse res )
                        throws ServletException, IOException {

        HtmlPage hp = new HtmlPage("Html page from main");

        // create four threads; each thread runs and completes before the next
        // thread is created.
        // MTHtml is a user-defined class. See below for the definition.
        for (int i=0; i<4; i++) {
          try {
             Runnable r = new MTHtml(hp); // user-defined class extending Thread
             Thread t = new WRBRunnable(r); // adopt r into the WRB context
             t.start(); // start the thread; this executes MTHtml's run() method
             t.join(); // wait for the thread to complete
          } catch (oracle.owas.wrb.WRBNotRunningException e) {
             System.err.println("The WRB is not running.");
          } catch (java.lang.InterruptedException e) {
             System.err.println("The sleeping thread was interrupted.");
          } // try
        } // for

        hp.printHeader();
        hp.print();
    } // doGet()
} // HelloThread class

class MTHtml extends Thread {
    private HtmlPage hp;

    public MTHtml(HtmlPage hp) {
        super();
        this.hp = hp;
    } // constructor

    public void run() {
        String thrName = Thread.currentThread().getName();
        HtmlBody hb = hp.getBody();
        hb.addItem("HelloWorld from " + thrName);
    } // run()
} // MTHtml class
```

The JServlet cartridge produces the following output:

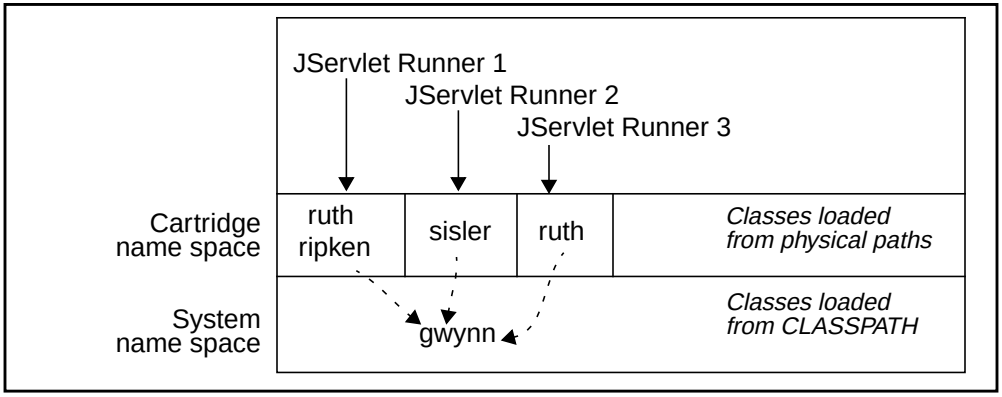
```
Content-type:    text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<!--  Generated by Oracle's Dynamic HTML Generation Package  -->
<HTML>
<HEAD>
<TITLE>Html page from main</TITLE>
</HEAD>
<BODY>
HelloWorld from Thread-5HelloWorld from Thread-7HelloWorld from
Thread-9HelloWorld from Thread-11</BODY>
</HTML>
```

Name Spaces of Java Classes

The name space of your Java classes depends on the location of the class. Figure 4–2 shows the different name spaces:

Figure 4–2 Name spaces of Java classes



When a JServlet cartridge loads a class from CLASSPATH, it loads it into the system name space. This means that all other classes, including those in the cartridge name space, can access it. In the figure above, the class gwynn was loaded from CLASSPATH.

When a JServlet cartridge loads a class from the physical path associated with the cartridge's virtual path, it loads the class into the name space of the cartridge.

Classes in the system name space and classes in other cartridges cannot access that class instance. In the figure above, the `ruth` and `ripken` classes can access each other in JServlet Runner 1, and access classes in the system name space (for example, `gwynn`), but they cannot access classes in the name spaces of other cartridges. For example, they cannot access `sisler` in JServlet Runner 2 or `ruth` in JServlet Runner 3.

When designing your application, consider what other classes your classes access. If you load a class into the system name space (that is, the class is found in CLASSPATH), then it cannot access classes that are found in physical paths. You should place your commonly accessed classes in CLASSPATH, and place only those classes to which you want to limit access in physical paths.

The following related topics are discussed in this section:

- Reflection APIs
- Using Packages
- Adding Classes with Native Libraries

Reflection APIs

Generally, classes in CLASSPATH cannot create new instances of or invoke methods on classes in physical paths.

However, if you use the Java Reflection APIs, you can invoke methods on classes in physical paths from classes in CLASSPATH. You cannot create new instances of classes in physical paths from classes in CLASSPATH.

The general steps to invoke methods on physical path classes from CLASSPATH classes are:

1. Get a reference to the class instance in physical path. The class instance must already exist.
2. Get the name of the method that you want to invoke and prepare the parameters for the method. This is an array of `Objects` passed to the `invoke()` method.
3. Invoke the method using `java.lang.reflect.Method.invoke()`.

In the following example, class `ruth` creates an instance of class `gwynn`, and the instance of class `gwynn` invokes a method on `ruth`. `ruth` is loaded from a physical path, while `gwynn` is loaded from CLASSPATH.

```
public class ruth {
    public void foo() {
        // create a new instance of fred
        gwynn f = new gwynn();
        // invoke a method on gwynn, and pass an instance of ruth
        f.doSomething(this);
    }
}

public class gwynn {
    public void doSomething(Object ruthInstance) {
        Class[] params = null;
        Class ruthClass = ruthInstance.getClass();
        Method homerun = ruthClass.getMethod("method_in_ruth", params);

        // invoke the method "method_in_ruth" which is in ruth
        homerun.invoke(ruthInstance, null);
    }
}
```

See the Java Reflection API documentation for details.

Using Packages

It is recommended that you place all your Java class files accessed by a JServlet cartridge directly in the physical path directory associated with the cartridge's virtual path. Using a flat directory structure allows the JServlet cartridge to load the classes in the directory into a private name space.

If you use custom packages, you have the following options:

- Make the package directory different from the physical path directory
- Make the package directory an immediate subdirectory of the physical path directory
- Make the package directory under the physical path directory, but with intermediate directories between the physical path and the package directory.

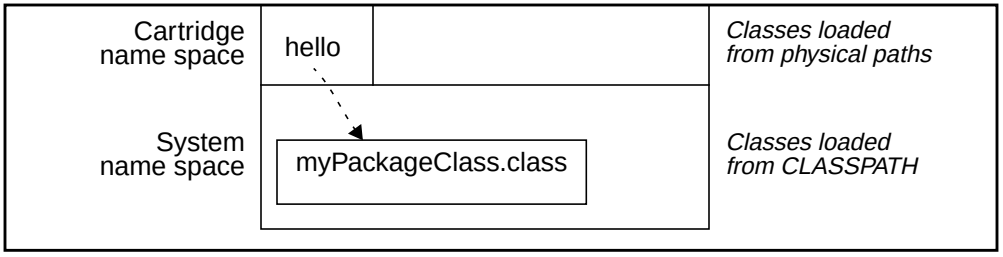
Package Directory Different From the Physical Path Directory

For example, your physical path is **c:\jservlets\myApp\cart1**, and your package directory is **c:\myjava\packages**. This enables you to place the package directory in the application's CLASSPATH, which means that the cartridge loads the package

classes into the system name space. Classes loaded from the physical path directory are still loaded in the JServlet runner's name space.

In the following figure, the cartridge loads the hello class from the physical path and places it in the JServlet runner name space, and loads a package class from the CLASSPATH and places it in the system name space.

Figure 4–3 Loading classes from a directory different from the physical path

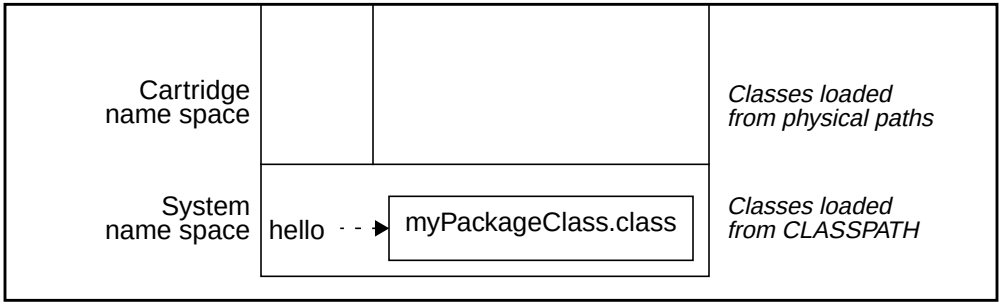


Package Directory in a Subdirectory Directly Under the Physical Path Directory

For example, your physical path is **c:\apps\jservlets\myApp\cart1**, and your package directory is **c:\apps\jservlets\myApp\cart1\myPackage**. In this case, to make the package classes accessible from your cartridge classes, you have to add **c:\apps\jservlets\myApp\cart1** to the CLASSPATH. This path is the same as the physical path, and this means that your cartridge classes and your package classes are loaded into the system name space.

In the following figure, the cartridge loads the hello class and a package class from the CLASSPATH and places them in the system name space.

Figure 4–4 Loading classes from a directory directly under the physical path

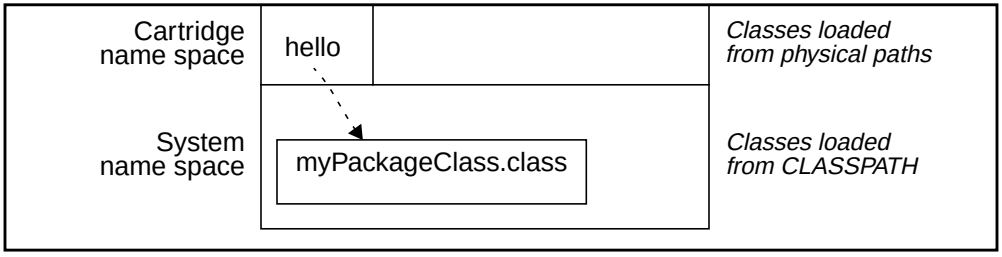


Package Directory Not Directly Under the Physical Path Directory

For example, your physical path is **c:\apps\jservlets\myApp\cart1**, and your package directory is **c:\apps\jservlets\myApp\cart1\someDirectory\myPackage**. In this case, to make the package classes accessible from your cartridge classes, you have to add **c:\apps\jservlets\myApp\cart1\someDirectory** to the CLASSPATH. This means that your cartridge classes are still loaded into the cartridge name space, but your package classes are loaded into the system name space.

In the following figure, the cartridge loads the hello class from the physical path and places it in the JServlet runner name space, and loads a package class from the CLASSPATH and places it in the system name space.

Figure 4–5 Loading classes from a directory not directly under the physical path



Adding Classes with Native Libraries

To add classes or packages with native libraries, the native libraries should be put in one of the directories listed in the LD_LIBRARY_PATH configuration setting of the application. For example, if you put your native libraries under the **/usr/local/oracle/java/lib** directory, your LD_LIBRARY_PATH should contain:

```
LD_LIBRARY_PATH = ...:/usr/local/oracle/java/lib:...
```

In the **wrb.app** file, the setting would look like:

```
[APPLICATION.<appName>.ENV]
...
LD_LIBRARY_PATH = ...:/usr/local/oracle/java/lib:...
```

5

Invoking Components

This chapter discusses invoking Oracle Application Server components through the JServlet applications.

Contents

Invoking ECO/Java Objects

Java applications running in a JServlet cartridge environment can invoke ECO / Java objects. ECO / Java objects are CORBA objects running in Oracle Application Server processes. See *Developer's Guide: EJB, ECO/Java and CORBA Applications* for more information on how to create ECO / Java objects and add them to the application server.

CLASSPATH

Before you can compile and run the Java application, add the following paths to the CLASSPATH of your development environment and to the CLASSPATH of the runtime environment of the Java application.

Development Environment

In your development environment, set CLASSPATH to contain:

- **$ORAWEB_HOME/classes/ecoapi.jar**

Note that you need to develop the ECO/Java application first before you can develop the JServlet client. This is because the client refers to the classes in the ECO/Java application.

If the ECO/Java objects throw user-defined exceptions, you have to include the class files for the exceptions in the development environment, because the client has to catch these exceptions.

Runtime Environment

In the runtime environment, set CLASSPATH (using the Environment Variables form for the JServlet application) to contain:

- **%ORAWEB_HOME%/classes/ecoapi.jar**
- **%ORACLE_HOME%/ows/apps/eco4j/<ECO/Java_appName>/_client.jar**
- The JAR file for your JServlet application. This JAR file contains the class files for your Java application and exception class files for exceptions thrown by the ECO/Java object. See *Developer's Guide: EJB, ECO/Java and CORBA Applications* for details on how to create these files.

  For the example below, the Jar file is called **Demo.jar**, and it contains:

```
> jar tvf Demo.jar
   599 Thu Jul 01 09:31:24 PDT 1999 META-INF/MANIFEST.MF
  3659 Thu Jul 01 09:31:22 PDT 1999 Demo.class
     0 Thu Jul 01 09:31:22 PDT 1999 myStack/
   259 Thu Jul 01 09:31:22 PDT 1999 myStack/HelloWorldException.class
   419 Thu Jul 01 09:31:22 PDT 1999 myStack/HelloWorldRemote.class
   315 Thu Jul 01 09:31:22 PDT 1999 myStack/HelloWorldHome.class
```

Entries in the CLASSPATH will be searched in order, so it is important that these CLASSPATH items be entered in the order given here.

Example

The following example shows a Java application called “Demo” invoking a ECO/Java object called “HelloWorld”.

Example 5–1 JServlet client for the ECO/Java server application

```
import myStack.HelloWorldRemote; // class for the ECO/Java object
import mystack.HelloWorldHome;
import oracle.oas.eco.PortableRemoteObject;
import javax.naming.*;
import javax.servlet.*;
import javax.servlet.http.*;

class Demo {
    public void doPost( HttpServletRequest req, HttpServletResponse res )
                        throws ServletException, IOException {
        res.setContentType(“text/html; charset=us-ascii”);
        Printwriter out = res.getWriter();
        HelloWorldHome s = null;

        try {
            // Initial Context
            Context _initialContext = new InitialContext();

            // Get an instance of a ECO/Java object named "HelloWorld" in a
            // ECO/Java application named "myStack"
            String _name = "oas:///myStack/HelloWorld";
            s = (HelloWorldHome) PortableRemoteObject.narrow(
                        _initialContext.lookup(_name),HelloWorldHome.class );

            // invoke a method called "helloWorld" on the ECO/Java object
            String result = s.helloWorld();
            out.println("Got back: " + result);
        } catch ( NamingException ne ) {
            System.err.print("Error finding server application");
        } catch ( org.omg.CORBA.SystemException se ) {
            System.err.print("Communication error.");
        } finally {
            if (s != null)
                s.destroy(); // from ECORemote
        }
    } // doPost
} // Demo
```

Example 5–2 ECO/Java server application

```
package myStack;
import oracle.oas.eco.*;
import javax.naming.InitialContext;

public class HelloWorld implements SessionBean {
    public String helloWorld() {
        return "Hello World, distributed style!";
    }
}
```

To run this application in a JServlet cartridge environment, the URL would look something like:

http://*machine*:*port*/**test/accessECO/Demo**

where *machine* is a listener running the JServlet cartridge, **/test/accessECO** is a virtual path for the JServlet cartridge, and Demo is the Java class that accesses the ECO / Java object.

Invoking Enterprise Java Beans

Similar to Invoking ECO / Java Objects, Enterprise Java Beans (EJBs) can be invoked with JServlet applications. The procedure and code are also similar. See *Developer's Guide: EJB, ECO/Java and CORBA Applications* for more information on how to create Enterprise Java Beans and add them to the application server.

CLASSPATH

Before you can compile and run the Java application, add the following paths to the CLASSPATH of your development environment and to the CLASSPATH of the runtime environment of the Java application.

Development Environment

In your development environment, set CLASSPATH to contain:

- **$ORAWEB_HOME/classes/ejbapi.jar**

Note that you need to develop the EJB application first before you can develop the JServlet client. This is because the client refers to the classes in the EJB application.

If the EJBs throw user-defined exceptions, you have to include the class files for the exceptions in the development environment, because the client has to catch these exceptions.

Runtime Environment

In the runtime environment, set CLASSPATH (using the Environment Variables form for the JServlet application) to contain:

- **%ORAWEB_HOME%/classes/ejbapi.jar**
- **%ORACLE_HOME%/ows/apps/ejb/<EJB_appName>/_client.jar**
- The JAR file for your JServlet application. This JAR file contains the class files for your Java application and exception class files for exceptions thrown by the ECO / Java object. See *Developer's Guide: EJB, ECO/Java and CORBA Applications* for details on how to create these files.

  For the example below, the Jar file is called **Demo.jar**, and it contains:

```
> jar tvf Demo.jar
   599 Thu Jul 01 09:31:24 PDT 1999 META-INF/MANIFEST.MF
  3659 Thu Jul 01 09:31:22 PDT 1999 Demo.class
     0 Thu Jul 01 09:31:22 PDT 1999 myStack/
   259 Thu Jul 01 09:31:22 PDT 1999 myStack/HelloWorldException.class
   419 Thu Jul 01 09:31:22 PDT 1999 myStack/HelloWorldRemote.class
   315 Thu Jul 01 09:31:22 PDT 1999 myStack/HelloWorldHome.class
```

Entries in the CLASSPATH will be searched in order, so it is important that these CLASSPATH items be entered in the order given here.

Example

The code in Example 5–1, "JServlet client for the ECO / Java server application" and Example 5–2, "ECO / Java server application" can be easily converted to support Enterprise Java Beans instead of ECO / Java. The following changes need to be made:

1. Import the necessary EJB classes for the client.

Replace the `import oracle.oas.eco.PortableRemoteObject` statement with:

```
import javax.rmi.PortableRemoteObject;
```

2. Catch the EJB RemoteException.

Replace the `org.omg.CORBA.SystemException` exception in the catch block with `java.rmi.RemoteException`.

3. Import the necessary classes for the EJB server application.

In the server application in Example 5–2, replace all of the existing import statements with:

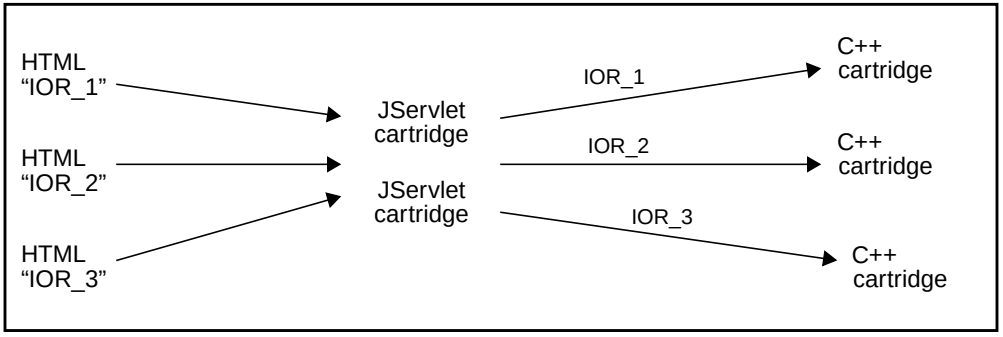
```
import javax.ejb.*;
import oracle.oas.ejb.*;
import javax.naming.*;
```

Invoking C++ CORBA Applications

Since C++ cartridges implement business logic, and typically do not specify how data is presented to clients. You can display data from C++ applications in browsers by using JServlet applications to format a C++ application's output.

The HTML page passes to the JServlet cartridge the IOR of the C++ cartridge with which it is communicating. The JServlet cartridge then invokes methods on the specified cartridge instance. (Figure 5–1 represents this graphically.) To store the IOR, the HTML page can store stringified IORs in hidden fields or it can use cookies.

Figure 5–1 HTML to JServlet cartridge to C++ cartridge



The request lifecycle for this task is:

1. A client sends the first request to the JServlet cartridge, which makes JNDI calls to get an object reference to a C++ cartridge.
2. The JServlet cartridge stringifies the IOR of the C++ cartridge, and returns an HTML page that includes the IOR value in a hidden field.
3. The user enters some data and submits the form. Note that the form should be submitted using the POST method (instead of GET) because of the long length of the stringified IOR.
4. The JServlet cartridge receives the form, which includes the stringified IOR, and invokes methods on the C++ cartridge referenced by the IOR.
5. Methods in the C++ cartridge return values to the JServlet cartridge, which assembles an HTML page using the values and returns it to the user. The JServlet cartridge can include the IOR on the HTML page if the user still needs to communicate with the C++ cartridge.
6. When no further communication is expected, the JServlet cartridge destroys the C++ cartridge.

You should set the timeout values for the C++ cartridges to conservative values because users can change their minds and not get to the point where the JServlet cartridge destroys the C++ cartridge. The timeout value ensures that Oracle Application Server cleans up idle, but occupied, C++ cartridges.

6

Database Access

JServlet applications can access databases in different ways:

- To invoke PL/SQL procedures and functions from Java applications running in the context of a JServlet cartridge, you can use the **pl2java** utility to generate Java wrapper classes for procedures in PL/SQL packages. You can then call the wrapper classes to invoke the PL/SQL procedures. This allows you to implement database logic in PL/SQL to ensure proper control of data in your databases and to invoke existing PL/SQL code from Java applications. See Chapter 7, "pl2java" for information on how to run the utility.
- To invoke SQL statements directly or access non-Oracle databases, you can use the JDBC package or JdbcBeans.

  JDBC provides a standard interface to access databases from different vendors. See "JDBC Example" on how to use JDBC with JServlet cartridges. For complete documentation on JDBC, see your vendor's documentation.

Contents

Using JDBC Drivers

The JServlet cartridge ships with **jts805jdbc.jar** and **oraclejts.jar** as the default JDBC and JTS libraries. The default behavior is for these drivers to be set in the CLASSPATH for any JServlet application. If JServlet detects the Oracle Application

Server runtime is an Enterprise Edition and transactions are enabled, the JServlet environment will bootstrap JTS. After that, any JServlet cartridge class can use JTS.

If you don't want to use JTS, you can remove these two jar files from the CLASSPATH and replace them with the regular Oracle JDBC jar file. In this case, you can use either the thin or thick JDBC driver. You also need to disable transactions using the OAS Manager.

By disabling transactions using the OAS Manager, you disable distributed transactions. Regular transactions supported by the JDBC drivers are still available.

Opening and Closing Connections

To improve performance it is best to open a database connection in a servlet's `init()` method and close the connection in the `destroy()` method. This can only be done if both:

- the servlet implements the `SingleThreadModel` interface
- an OCI driver is used.

If either condition is not met, the database connection must be opened and closed within the same method.

Error Handling

If the database connection is lost while a servlet instance is still alive, attempting to use the connection will raise the `SQLException` exception. If the connection is not re-established, all requests sent to this servlet instance will fail. Additional error checking should be added to check for connection failure and perform appropriate exception handling. This will make your applications more robust.

Using Multibyte Character Sets

If you want to use the JDBC OCI8 Driver to handle multibyte characters from a database, the character set must be set to UTF8 in the Java Environment form. For example, to use the Japanese character set, define the NLS_LANG parameter as `NLS_LANG=JAPANESE_JAPAN.UTF8`.

JDBC Example

The JServlet cartridge has been tested with Oracle's JDBC OCI7 and OCI8 driver to connect to Oracle databases. For details on the driver, see the JDBC documentation.

Note that you have to include the following paths to LD_LIBRARY_PATH and CLASSPATH in order to compile and run the Java application.

- LD_LIBRARY_PATH: include **%ORACLE_HOME%/jdbc/lib**. %ORACLE_HOME%/jdbc is the JDBC root directory.
- CLASSPATH: include **%ORACLE_HOME%/jdbc/lib/classes111.zip**.

You add these paths to the application using the Environment Variables form in the Oracle Application Server Manager.

Example 6–1 Using the JServlet toolkit with JDBC

```
// You need to import the java.sql package to use JDBC
import java.sql.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import oracle.html.*;  // import Oracle HTML classes which deal with HTML

public class JDBC_EmployeeReport extends HttpServlet
                                    implements SingleThreadModel {
  private Connection conn = null;

  public void init( ServletConfig config ) throws ServletException {
    super.init(config);
    try {
       // Load the Oracle JDBC driver
       Class.forName ("oracle.jdbc.driver.OracleDriver");

       // Connect to the database. To connect to a remote database,
       // insert the connect string after the @ sign in the connection URL.
       Connection conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                                     "www_user","manager");
    } catch ( SQLException e ) {
       System.err.println("Could not establish connection.");
    } catch ( ClassNotFoundException e ) {
       System.err.println("Could not load database driver.");
    }
  } // init
```

```
public void doPost( HttpServletRequest req,
                    HttpServletResponse res )
                    throws ServletException, IOException {
  try {
     // Create a Statement
     Statement stmt = conn.createStatement ();

     // Select the ENAME column from the EMP table
     ResultSet rset = stmt.executeQuery("select ename, empno, deptno from EMP");

     // create an HTML page to return to the client browser
     HtmlHead hd = new HtmlHead("Employee Listing");
     HtmlBody bd = new HtmlBody();
     HtmlPage hp = new HtmlPage(hd, bd);

     // generate report
     DynamicTable tab = new DynamicTable(2);
     TableRow row = new TableRow();
     row.addCell(new TableHeaderCell("Employee Name"))
        .addCell(new TableHeaderCell("Employee Number"))
        .addCell(new TableHeaderCell("Employee Dept"));
     tab.addRow(row);
     // Iterate through the result and print the employee names
     while ( rset.next () ) {
        row = new TableRow();
        if ( rset.getInt(2)==0 ) {
           row.addCell(new TableDataCell(rset.getString(1)))
               .addCell(new TableDataCell("new employee"))
               .addCell(new TableDataCell(rset.getString(3)));
        } else {
            row.addCell(new TableDataCell(rset.getString(1)))
               .addCell(new TableDataCell(String.valueOf(rset.getInt(2))))
               .addCell(new TableDataCell(rset.getString(3)));
        }  // if
        tab.addRow(row);
     } // while
     hp.addItem(tab);
     hp.printHeader();
     hp.print();
  } catch ( SQLException e ) {
     System.err.println("A database error occurred.");
  }
} // doPost
```

```
  public void destroy() {
    try {
       // close the database connection
       if ( conn != null) conn.close();
    } catch (SQLException e) {
       System.err.println("Error closing database connection.");
    }
  } // destroy

} // class
```

Using the Transaction Service

> **Note:** This feature is available in the Enterprise Edition of Oracle Application Server only.

You can use Oracle Application Server's transaction service with the JServlet cartridge. The transaction service works with different database access APIs to coordinate distributed transactions. In the case of the JServlet cartridge, the database access APIs are JDBC (Java Database Connectivity) and **pl2java**-generated classes. You use the database access APIs to perform database operations such as connecting to the database, executing statements on the database, and disconnecting from the database. You do not use the database access APIs to demarcate transactions; instead, you use the transaction service APIs.

The transaction service enables you to perform transactions that span requests, resource managers, and cartridges. See the "Enabling Transactions," chapter in the Administrator's Guide for information on transactions and on configuring transactions.

The transaction service is based on JTS (Java Transaction Service). You can get more information about this service from the JavaSoft site (**http://java.sun.com/products/jts**).

When the JServlet cartridge starts up, it calls `initTS()` to initialize the service. You should not call `initTS()`. When it shuts down, it calls `termTS()` to clean up the service.

Transaction Service with JDBC

To use the transaction service with JDBC in the JServlet cartridge:

1. Connect to a resource manager using `TransactionService.connectRM()`. The method takes one parameter specifying the name of a transactional DAD. The connectRM() method is part of the oracle.jts.util.TransactionService class. You may retrieve the TransactionService object through the static method, TS::getTS.
2. Get an `org.omg.CosTransactions.Current` object using `TransactionService.getCurrent()`. The Current object provides the `begin()`, `commit()`, and `rollback()` methods to demarcate transactions.
3. Begin a transaction through the `begin()` method.
4. Connect to the database using `DriverManager.getConnection()`.
5. Perform database operations.
6. Commit or roll back the transaction through either the `commit()` or `rollback()` methods.
7. Disconnect from the resource manager using `disconnectRM()`.

Example 6–2 Transaction service with JDBC

```
// A JTS/JDBC test program; single process single-threaded test.
// It will require you to have a database, orb, and otsfacsrv running.

import java.sql.*;
import java.math.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Allows us to refer to the TS class without oracle.jts.
import oracle.jts.util.*;
// Allows us to refer to Current and Status without prefix.
import org.omg.CosTransactions.*;

// A basic test program which exercises JTS.
// This program is not meant to actually do anything useful,
// except exercise the basic functionality of JTS.
public class testjts {
  // Note: The name of the transactional DAD is hard-coded here. The variable
  // TheTestRM is set to the DAD name "testrm1". You need a transactional
```

```
// name of the same name, or you will need to modify the DAD name here.
String TheTestRM = "testrm1"; // the name of a transactional DAD
String TheTestDB = "otsinst1";

// This method must contain all JTS actions for this thread.
// This method is called within the Transactional.run() method.
public void TheTest() {
  try {
     (TS.getTS()).connectRM(TheTestRM);
  } catch (Exception e) {
    System.err.println("testjts: Connect failed: message="+e.getMessage());
    return;
  }

  // Get a Current object
  Current mycurrent = (TS.getTS()).getCurrent();

  // Now exercise the Current object
  // Begin() starts a transaction
  ResultSet rset,r2set;
  Statement stmt;
  Connection conn = null;

  try {
    mycurrent.begin();
    mycurrent.set_timeout(10);
    Status ourstat = mycurrent.get_status();

    String traname = mycurrent.get_transaction_name();
    System.err.println("testjts: Transaction_name="+traname);

    try {
      // Do actual JDBC here
      DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
      System.err.println("testjts: after registerDriver");

      // Load the JTS enabled JDBC driver; login to db
      conn = DriverManager.getConnection("jdbc:oracle:jts7:@"+TheTestRM,
                                         "tkxabrch", "tiger");
      System.err.println("testjts: after getConnection");
      stmt = conn.createStatement();
      System.err.println("testjts: after createStatement");
      try {
        conn.setAutoCommit(true);
      } catch (Exception e) {
```

```
  System.err.println("testjts: Correct error: setAutoCommit failed:"
                     +e.getMessage());
}

System.err.println("testjts: Before: names in the database:");
r2set = stmt.executeQuery("select ename from emp");
while(r2set.next()) {
  System.err.println("testjts:  Name="+r2set.getString(1));
}

System.err.println("testjts: Doing insert.");
int numrows = stmt.executeUpdate(
  "insert into emp(empno,ename,sal) VALUES (2003,'JunIsKing',912)");

System.err.println("testjts: Doing insert.");
numrows = stmt.executeUpdate(
  "insert into emp(empno,ename,sal) VALUES (2004,'Ziggy',912)");

System.err.println("testjts: after inserts: names in the database:");
r2set = stmt.executeQuery("select ename from emp");
while(r2set.next()) {
  System.err.println("testjts:  Name="+r2set.getString(1));
}

System.err.println("testjts: Doing delete.");
numrows=stmt.executeUpdate("delete from emp where ename='JunIsKing'");

System.err.println("testjts: Names before commit.");
r2set = stmt.executeQuery("select ename from emp");
while(r2set.next()) {
  System.err.println("testjts: Name="+r2set.getString(1));
}

System.err.println("testjts: Trying a commit!");
mycurrent.commit(false);

try {
  System.err.println("testjts: names in database afterwards.");
  r2set = stmt.executeQuery("select ename from emp");
  while(r2set.next()) {
    System.err.println("testjts:  Name="+r2set.getString(1));
  }
} catch (Exception e) {
  System.err.println("testjts: Correct exception: tran-less sql: "
        +e.getMessage());
```

```
      }

    } catch (SQLException esqlerr) {
      System.err.println("testjts: SQL Exception:"+esqlerr.getMessage());
    } catch (Exception except) {
      System.err.println("testjts: General Exception: "+except.getMessage());
    }

    try {
      System.err.println("testjts: Verifying get_transaction_name complains
              if no transaction.");
      String traname2 = mycurrent.get_transaction_name();
      System.err.println("testjts: You should never see me!");
    } catch (Exception e) {
      System.err.println("testjts: Correct exception: couldn't get
              transaction name");
    }

  } catch(Exception e) {
    System.err.println("testjts: Begin/Rollback exception: "+e.getMessage());
  }

  // Close the JDBC connection and the JTS connection
  try {
    if (conn != null) conn.close();

    (TS.getTS()).disconnectRM(TheTestRM);
  } catch (Exception e) {
    System.err.println("testjts: Disconnect failed: "+e.getMessage());
    return;
  }

  // finish!
  System.err.println("testjts: We are done!");
} // TheTest

// Lets that object run.
public void dothis() {
  int didwhat = 0;

  try {
    didwhat = 1;

    // And let it run!
    TheTest();
```

```
      didwhat = 0;
    } catch (Exception e) {
      System.err.print("testjts: Caught exception! dowhat="+didwhat);
      System.err.println(" Message= " + e.getMessage());
    } // dothis

  // This test's entry point.
  // Creates a real object and lets it take over.
  public void doPost( HttpServletRequest req,
                      HttpServletResponse res )
                      throws ServletException, IOException {
    testjts athing = new testjts();
    athing.dothis();
  } // doPost
} // class
```

7

pl2java

This chapter describes how to use the **pl2java** utility to access databases. The **pl2java** utility generates Java wrapper classes for PL/SQL procedures and functions in PL/SQL packages.

You can then call the wrapper classes from your Java applications to invoke the PL/SQL subprograms. This allows you to implement database logic in PL/SQL to ensure proper control of data in your databases and to invoke existing PL/SQL code from Java applications.

Contents

Overview of pl2java

To enable Java applications to invoke PL/SQL procedures and functions, you can use **pl2java**, a utility that generates Java wrapper classes for PL/SQL stored procedures. A Java wrapper class is a class containing methods to call a PL/SQL package's procedures and functions, and serves as an interface between the two languages. Stand-alone PL/SQL procedures and functions are all wrapped in a single wrapper class.

The Java wrapper classes generated by **pl2java** provide a convenient way to invoke PL/SQL stored procedures. For each PL/SQL package you specify, **pl2java** generates a Java wrapper class, which contains a wrapper method for each procedure or function in the PL/SQL package. The prototype of the wrapper method is the same as the PL/SQL procedure or function it wraps. This provides an "object" view of PL/SQL packages and allows PL/SQL stored procedures to be called seamlessly.

pl2java can be found in the **$ORACLE_HOME/ows/cartx/jweb/bin** directory.

Requirements

To use **pl2java**, you must have:

- installed the packages in the database that you want to invoke from Java
- installed the Java Development Kit (JDK) version 1.1.6
- placed the Java interpreter executable in your execution path
- installed the `dbms_package` PL/SQL package in the database where your PL/SQL packages are loaded. This package must be installed by the SYS database user.

  For Oracle7, the installation script is **$ORACLE_HOME/ows/cartx/jweb/sql/dbpkins.sql**. The de-installation script is **dbpkdins.sql**.

  For Oracle8, use **dbpkins8.sql** instead of **dbpkins.sql**. If you use the wrong installation script, you will get the following error message:

  ```
  Warning: Package Body created with compilation errors
  ```

If any errors occur during the installation procedure, the **dbpkdins.sql** script should be run to undo any changes made.

Running pl2java

To generate the Java wrapper class for your PL/SQL package, invoke **pl2java** from the command prompt:

```
prompt> pl2java [-help] [-d dir] [-package pkg] [-class class] [-nolog]
                    username/password[@connect-string] plsql_package...
```

pl2java creates a Java wrapper class for each PL/SQL package given as an argument to the command. When your application is run, an instance of this class to interface to the package is created.

If you have stand-alone procedures or functions in your applications, run **pl2java** with the `class` flag. This creates a single wrapper class for all the stand-alone procedures and functions you use.

The following table lists the arguments of **pl2java**:

Table 7–1 Arguments for pl2java

Argument	Description
-help	Provides help information
-d *dir*	The directory where the wrapper classes will be stored; the default is the current directory.
-package *pkg*	The Java package to which the wrapper classes belong.
-class *class*	The Java class to which the wrappers belong. If the **pl2java** utility is run against packages, this flag is optional. Java classes based on packages inherit by default the names of the packages they encapsulate. This flag can override the default, but it only applies to the first package named in the command. If the wrappers are being created for stand-alone procedures and functions, then this flag is mandatory, and all procedures and functions named in the command are grouped into the single class named by this flag.
-nolog	If this option is specified, the log messages (such as "Connecting to database" and "Username is *user*") produced by the generated classes are not logged. If this option is not specified (the default), the log messages are written to the log device (such as a file or database), if logging is enabled for the application using the generated classes.
username	The name of the Oracle database user that owns the PL/SQL packages
password	The password for the Oracle user identified by username

Table 7–1 Arguments for pl2java

Argument	Description
connect-string	The string that identifies the database where the packages are located. This is the SQL*Net Connect String, as described in Understanding SQL*Net. For local databases, omit this connect-string. Instead, set the `ORACLE_SID` environment variable to specify the local databases.
plsql_package...	A list of all the PL/SQL packages that your Java application references in the schema identified by username. To wrap stand-alone procedures and functions, omit this argument and instead use the `-class` flag to name the class wrapper that will be created. You should not include the containing schemas in the package names. It is good practice to keep all the packages, procedures, and functions you want to use in one schema.

The names of the classes follow the capitalization given in the command. However, since PL/SQL is not case-sensitive, this capitalization does not need to follow the capitalization in the PL/SQL code itself.

PL/SQL Data Type Mapping in Java

To pass data between Java and PL/SQL, **pl2java** maps the PL/SQL data types to Java wrapper classes. Primitive types in Java cannot represent PL/SQL types because PL/SQL types can have null values, which are not allowed in Java. So to represent PL/SQL types without losing the null value, **pl2java** maps PL/SQL types to Java wrapper classes.

These wrapper classes belong to the `oracle.plsql` package and are derived from the `PValue` base class, which encapsulates the null attribute of PL/SQL values. Each individually derived wrapper class represents one or more related PL/SQL data types. The following table lists the Java wrapper classes for PL/SQL data types:

Table 7–2 Wrapper classes for PL/SQL data types

PL/SQL data type	Java wrapper class
BINARY_INTEGER (NATURAL)	PInteger
NUMBER (DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, INT, NUMERIC, REAL, SMALLINT)	PDouble
CHAR(n) (CHARACTER, STRING)	PStringBuffer

Table 7–2 Wrapper classes for PL/SQL data types

PL/SQL data type	Java wrapper class
VARCHAR2(n) (VARCHAR)	PStringBuffer
LONG	PStringBuffer
RAW (n), LONG RAW	PByteArray
LONG RAW	PByteArray
BOOLEAN	PBoolean
DATE	PDate
PL/SQL table	Java array

When you pass a value to a PL/SQL call, you create a wrapper object for that PL/SQL data type and store the value in it. If the call returns with an output parameter, you retrieve the value from the wrapper object.

There are certain PL/SQL data types that **pl2java** cannot encapsulate. These are shown below, along with the recommended substitutes, if any:

Table 7–3 Unsupported PL/SQL data types

Disallowed PL/SQL data type	Substitute PL/SQL data type
POSITIVE	BINARY INTEGER
CLOB, BLOB, BFILE	none
PL/SQL table of BINARY INTEGER, NATURAL or POSITIVE	PL/SQL table of NUMBER
PL/SQL table of LONG	PL/SQL table of CHAR or VARCHAR2
PL/SQL table of BOOLEAN	PL/SQL table of NUMBER, treat 0 as false, 1 as true
ROWID	none
MSLABEL	none
PL/SQL table of ROWID	none
PL/SQL table of MSLABEL	none

Records and cursors are not supported.

Example

Consider the following Employee PL/SQL package that contains a function and a procedure:

Example 7–1 Employee PL/SQL package

```
package Employee as
   type string_table is table of varchar2(30) index by binary_integer;
   type number_table is table of number(10) index by binary_integer;
   function count_employees(dept_name in varchar2) return number;
   procedure list_employees(
      dept_name          in    varchar2,
      employee_name      out   string_table,
      employee_no        out   number_table
   );
end;
```

Run **pl2java** to generate the wrapper class:

```
prompt> pl2java scott/tiger@db Employee
```

pl2java generates the `Employee` wrapper class, which contains the following:

```
public class Employee {
   public Employee(Session dbSession) { ... }
   public PDouble count_employees(PStringBuffer dept_name) { ... }
   public void list_employees(
      PStringBuffer  dept_name,
      PStringBuffer  employee_name[],
      PDouble        employee_no[]
   ) { ... }
}
```

When a PL/SQL function returns a value whose size is variable (for example VARCHAR2, LONG, RAW, or LONG RAW), the size of the value is set by default to 255 bytes. In the wrapper class, you may change the default size by setting the following data member of the wrapper class for the PL/SQL function in question:

```
<function name>_<overload number>_return_length
```

The overload number is the number of other functions that exist with the same name. You can find the overload number of a function by using the Oracle Server standard package `DBMS_DESCRIBE`. For non-overloaded functions, the overload number is 0.

For example, assume the following PL/SQL package:

```
package Employee as
   function employee_name (
      employee_number    in      number
   ) return varchar2;
end;
```

The wrapper class Employee contains the data member `employee_name_0_return_length` which can be overridden:

```
public class Employee {
   ...
   public PStringBuffer employee_name(PDouble employee_number);
   public int employee_name_0_return_length = 255;
}
```

Similarly, if the function returns a PL/SQL table, you can specify the length of the array with the data member of the wrapper class:

```
<function name>_<overload number>_return_arraylength
```

Connecting to the Database

The database connection is encapsulated by the `Session` class in the `oracle.rdbms` package. The `Session` class provides methods to perform common database tasks, such as logon, logoff, commit, rollback, and so on. Before you connect to an Oracle database, you need to define environment properties, such as the ORACLE_HOME environment variable. To do this, retrieve ORACLE_HOME of the application server in the cartridge's system properties "`oracleHome`" with the `System.getProperty` method. After defining Oracle environment properties, instantiate a Session object and log on to the database. The following sample code illustrates how this is done:

Example 7–2 Connecting to a database

```
// Define ORACLE_HOME
Session.setProperty("ORACLE_HOME", System.getProperty("oracleHome"));

// Create a new database session and logon
Session session = new Session();
session.logon("scott", "tiger", "sales_db");
```

Invoking PL/SQL Stored Procedures

To invoke a PL/SQL stored procedure, you instantiate the wrapper class for the PL/SQL package (or for anonymous/stand-alone PL/SQL procedures) with a Session object. This prepares the object for the execution of the PL/SQL package in the session. If you want to invoke the PL/SQL package in multiple database sessions, you need to instantiate the wrapper class for each one of them. For example, to instantiate an Employee object:

```
// Instantiate Employee wrapper class:
Employee emp = new Employee(session);
```

When you invoke a PL/SQL procedure that takes parameters, you need to pass the values to the parameters by storing them in PL/SQL data type wrapper objects. Instantiate these wrapper objects and set the values with the `setValue` method and pass these objects to the wrapper method as parameters. To pass a PL/SQL null value, use `setNull` method on the wrapper object to set the value to null.

When a PL/SQL function returns, it may return some values in its out parameters or return value. To retrieve the return values, use the get-value methods of the wrapper classes. For example, use the `intValue` method of the `PInteger` class to retrieve an int value. Note that when a PL/SQL return value is null, the get-value method throws a `NullValueException`. This `NullValueException` signifies a null value, and you should handle this exception properly with a try-catch block. Alternatively, you can first use the `isNull` method to determine if the value is null, and invoke the get-value method when it is not null.

Example 7–3 Passing PL/SQL parameters in and out

```
// Instantiate a PStringBuffer to pass a string to the PL/SQL procedure
PStringBuffer pDeptName = new PStringBuffer(30);
pDeptName.setValue("Sales");

// Invoke the PL/SQL procedure
PDouble pEmployeeCount = employee.count_employees(pDeptName);

// Retrieve the return value
if (!pEmployeeCount.isNull())
   int employeeCount = pEmployeeCount.intValue();
```

When a PL/SQL parameter is a PL/SQL table, either an in or an out parameter, you need to create the Java array as well as the elements in the array. This is illustrated in Example 7–4.

Example 7–4 Passing a PL/SQL table in and out

```
// Create a PL/SQL table parameter
PStringBuffer pEmployeeNames[] = new PStringBuffer[30];
for(int = 0; i < pEmployeeNames.length; i++)
   pEmployeeNames[i] = new PStringBuffer(80);

// Invoke a PL/SQL procedure
employee.list_employees(pEmployeeNames);
```

All wrapper classes that encapsulate PL/SQL values have a `toString` method and therefore can be concatenated with Java Strings. For example, you can use the pEmployeeCount object from above directly in a string concatenation:

```
// Display the employee count
System.out.print("There are " + pEmployeeCount + " employees.");
```

Handling Database Errors

pl2java and the JServlet cartridge provide tight integration between Java and PL/SQL in the way database exceptions are handled. When an error occurs during a database operation, an exception is thrown. The exception is returned to Java and is thrown as a `ServerException`. For example, when a no-data-found exception occurs in a SQL `select` statement, a `ServerException` is thrown. You can use the `getSqlcode` and `getSqlerrm` methods to retrieve the SQL code and error message of the exception.

Most methods of the `Session` class as well as the wrapper methods in PL/SQL wrapper classes throws `ServerException`. You should catch the exception by putting the calls in a try-catch block.

Setting the Character Set Value

The character set that an application uses when communicating with a database is specified by the NLS_LANG variable set in the Environment Variables form in the Oracle Application Server Manager. The value of the variable is used by all cartridges in the application. To use more than one NLS_LANG value, you need to create separate applications.

If the NLS_LANG value is not specified for an application, it defaults to `AMERICAN_AMERICA.US7ASCII`. See the Oracle Server documentation for the format of NLS_LANG.

Note: You should not change NLS_LANG programmatically because the value must be defined before the cartridge server starts up. If you use the `Session.setProperty()` method to set NLS_LANG, the first request to the cartridge will fail due to NLS bootstrapping limitations and may produce undesired behavior.

Freeing Database Sessions

Java provides a garbage collector which frees up objects when they are no longer needed. When a database session is no longer needed and becomes garbage, the session will be disconnected before it is garbage-collected. However, the garbage collector does not guarantee that any garbage objects will be collected immediately. In fact, Java's garbage collector waits until the program is idle, or until resources are low, before it collects garbage objects. Therefore, you should try to log off from the database when the session is no longer needed to free up database resources explicitly.

Using the Transaction Service with pl2java

Note: This feature is available in the Enterprise Edition of Oracle Application Server only.

You can use Oracle Application Server's transaction service with the **pl2java**-generated classes. The transaction service enables you to perform transactions that span requests, resource managers, and cartridges. See the *Administration Guide* for details.

The transaction service is based on JTS (Java Transaction Service). You can get more information about this service from the JavaSoft site (**http://www.javasoft.com**).

When the JServlet cartridge starts up, it calls `initTS()` to initialize the service. You should not call `initTS()`. When it shuts down, it calls `termTS()` to clean up the service.

This section contains the following subsections.

- Configuration
- Transaction Service with pl2java-generated Classes

Configuration

To use the transaction service for a JServlet application, check that you have done the following:

- Enabled the transaction service for the application. You do this using the Transaction Property form.
- Specified transactional DADs in the Transaction Property form. You use the DAD Transactions form to choose a transactional DAD.

Transaction Service with pl2java-generated Classes

To use the transaction service with **pl2java**-generated classes in the JServlet cartridge:

1. Connect to a resource manager using `connectRM()`. The method takes one parameter specifying the name of a transactional DAD.
2. Register a transactional DAD with **pl2java** classes using Session.
3. Get a Current object using `getCurrent()`. The Current object provides the `begin()`, `commit()`, and `rollback()` methods to demarcate transactions.
4. Begin a transaction.
5. Perform database operations.
6. Commit or roll back the transaction.
7. Disconnect from the resource manager using `disconnectRM()`.

Example 7–5 Transaction service with pl2java generated classes

```
import javax.servlet.*;
import javax.servlet.http.*;
import oracle.rdbms.*; // import Oracle classes that deal with database session
import oracle.plsql.*; // import Oracle classes that deal with PL/SQL data types
import oracle.html.*;  // import Oracle HTML classes that deal with HTML
import oracle.jts.util.*;
import org.omg.CosTransactions.*;

public class TXNEmployeeReport {
    public void doPost( HttpServletRequest req,
                        HttpServletResponse res )
                        throws ServletException, IOException {

    HtmlHead hd = new HtmlHead("Employee Listing");
```

```
HtmlBody bd = new HtmlBody();
HtmlPage hp = new HtmlPage(hd, bd);
hp.printHeader();

// define Oracle session properties link ORACLE_HOME
Session.setProperty("ORACLE_HOME", System.getProperty("oracleHome"));

Session session;
TXNEmployee employee;
String deptName;

// STEP 1. Connect to a resource manager
try {
    (TS.getTS()).connectRM("TXN"); // TXN is a transactional DAD
} catch (Exception e) {
    System.out.println("testpl2javatx: Connect failed: message="
                       + e.getMessage());
    return;
}

// STEP 2. Register the connection with PLSQL classes
try {
    session = new Session("TXN");
    // create a new instance of TXNEmployee package
    employee = new TXNEmployee(session);

    // find the department name from the input parameter
deptName = getArgument(args, "DEPT");
} catch (ServerException e) {
    bd.addItem(new SimpleItem("PLSQL Instance creation failed : "
                              + e.getSqlerrm()));
    hp.print();
    return;
}

try { // STEP 3. Get a current object.
  Current mycurrent = (TS.getTS()).getCurrent();
  mycurrent.begin();

  // STEP 4. Begin a transaction.
  generateReport(employee, deptName, bd, hp, "before adding new Employee");

  PStringBuffer psbadd = new PStringBuffer(30, "Hello");
  PStringBuffer pDeptName = new PStringBuffer(30, deptName);
  PDouble pdadd = new PDouble(1010);
```

```
  // STEP 5. Perform database operations.
  employee.add_employee(psbadd, pdadd, pDeptName);

  // STEP 6. Commit the transactions.
  mycurrent.commit(true);

} catch (Exception e) {
    e.printStackTrace(System.out);
}

try { // begin another transaction
  Current mycurrent = (TS.getTS()).getCurrent();
  mycurrent.begin();

  generateReport(employee, deptName, bd, hp, "after adding a new Employee");
  PStringBuffer psbdelete = new PStringBuffer(30, "Thomas Bird");
  employee.delete_employee(psbdelete);
  generateReport(employee, deptName, bd, hp, "after deleting an Employee");
  mycurrent.rollback(); // roll back the transaction
} catch (Exception e) {
    e.printStackTrace(System.out);
}

try {
  Current mycurrent = (TS.getTS()).getCurrent();
  mycurrent.begin();

  generateReport (employee, deptName, bd, hp, "after rollback on deletion");
  PStringBuffer psbdelete = new PStringBuffer(30, "Thomas Bird");
  employee.delete_employee(psbdelete);
  generateReport (employee, deptName, bd, hp, "after deleting an Employee");
  mycurrent.commit(true);
} catch (Exception e) {
    e.printStackTrace(System.out);
}

try {
  Current mycurrent = (TS.getTS()).getCurrent();
  mycurrent.begin();

  generateReport (employee, deptName, bd, hp, "after commit on deletion");
  mycurrent.commit(false);
} catch (Exception e) {
    e.printStackTrace(System.out);
```

```
    }

    // Deregister from PLSQL classes
    try {
        session.logoff();
    } catch (ServerException e) {
    }

    // STEP 7. Disconnect from the resource manager.
    try {
      (TS.getTS()).disconnectRM("TXN");
    } catch (Exception e) {
      System.out.println("testpl2javatx: Disconnect failed: " + e.getMessage());
      return;
    }
    hp.print();
} // doPost


// Look up a URL parameter
private static String getArgument(String args[], String name) {
    String prefix = name + "=";
    for(int i = 0; i < args.length; i++)
        if (args[i].startsWith(prefix))
            return args[i].substring(prefix.length());
    return null;
} // getArgument


private static void generateReport(TXNEmployee employee, String deptName,
                                   HtmlBody bd, HtmlPage hp, String label)
                    throws HtmlException {
    if ((deptName == null) || (deptName.length() == 0)) {
        bd.addItem(new SimpleItem("No department name given"));
        return;
    }

    // create objects to encapsulate PL/SQL values that are
    // used as parameters
    PStringBuffer pDeptName = new PStringBuffer(30, deptName);
    PStringBuffer pEmployeeName[];
    PDouble       pEmployeeNumber[];
    PDouble       pEmployeeCount;

    // print report header
```

```
bd.addItem(SimpleItem.Paragraph)
  .addItem("Employees from Department " + pDeptName + " " + label)
  .addItem(SimpleItem.Paragraph);

// call TXNEmployee package to count the number of employees in
// the department
try {
    pEmployeeCount = employee.count_employees(pDeptName);
} catch (ServerException e) {
    bd.addItem("Fail to retrieve employee information for department "
               + deptName + ": " + e.getSqlerrm());
    return;
}

int employeeCount = (int)pEmployeeCount.doubleValue();
if (employeeCount == 0) {
    bd.addItem("No employee found under department " + deptName);
    return;
}

// allocate the arrays for employee names and numbers
pEmployeeName   = new PStringBuffer[employeeCount];
pEmployeeNumber = new PDouble[employeeCount];

// allocate the buffers to retrieve employee information
for(int i = 0; i < employeeCount; i++) {
    // max length of employee name is 30 (characters)
    pEmployeeName[i]   = new PStringBuffer(30);
    pEmployeeNumber[i] = new PDouble();
}

// call TXNEmployee package to look up employees in the dept
try {
    employee.list_employees(pDeptName, pEmployeeName, pEmployeeNumber);
} catch (ServerException e) {
    bd.addItem("Fail to retrieve employee information for department "
               + deptName + ": " + e.getSqlerrm());
    return;
}

// generate report
DynamicTable tab = new DynamicTable(2);
TableRow row = new TableRow();
row.addCell(new TableHeaderCell("Employee Name"))
   .addCell(new TableHeaderCell("Employee Number"));
```

```
        tab.addRow(row);

        for (int i = 0; i < employeeCount; i++) {
            row = new TableRow();
            if (pEmployeeNumber[i].isNull())
                row.addCell(new TableDataCell(pEmployeeName[i].toString()))
                   .addCell(new TableDataCell("new employee"));
            else
                row.addCell(new TableDataCell(pEmployeeName[i].toString()))
                   .addCell(new TableDataCell(pEmployeeNumber[i].toString()));
            tab.addRow(row);
        }
        hp.addItem(tab);
        } // generateReport

    } // class TXNEmployeeReport
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W